České vysoké učení technické v Praze
Fakulta elektrotechnická

katedra počítačů

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student: **Bc. Karel Horák**

Studijní program: Otevřená informatika
Obor: Umělá inteligence

Název tématu: **Dotazovací engine grafové databáze využívající stromové dekompozice**

Pokyny pro vypracování:

1) Proveďte rešerši literatury týkající se homomorfismu grafů.
2) Vytvořte prototyp dotazovacího stroje pro grafovou databázi, který využívá stromovou dekompozici dotazu pro efektivní vyhodnocování.
3) Výkonnost ověřte empiricky na vybraných datasetech.

Seznam odborné literatury:

- Yannakakis, Algorithms for acyclic database schemes, 1981
- Beeri, Properties of acyclic database schemes, 1981
- Blair and Peyton, An introduction to chordal graphs and clique trees, 1992
  Grohe et al., When is the evaluation of conjunctive queries tractable?, 2001
- Flum et al, Query Evaluation via Tree-Decompositions, 2002
- Grohe, The Complexity of Homomorphism and Constraint Satisfaction Problems Seen from the Other Side, 2007
- Bodlaender et al., A c^k*n 5-Approximation Algorithm for Treewidth, 2013

Vedoucí: MSc. Radomír, Černoch

Platnost zadání: do konce letního semestru 2015/2016

L.S.

doc. Ing. Filip Železný, Ph.D.
vedoucí katedry

prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 26. 3. 2015

Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Computer Science and Engineering

Diploma Thesis

Karel Horák

Graph database query engine based on tree decompositions

May 2015

Supervisor: Radomír Černoch, MSc.

Acknowledgement

I would like to express my gratitude to my supervisor Radomír Černoch, MSc. for the time he spent with me discussing the topic, his patience and all his advice that helped me to stay on track during the whole period of writing this thesis.

I would like to thank my family and friends for their support.

Access to computing and storage facilities owned by parties and projects contributing to the National Grid Infrastructure MetaCentrum, provided under the programme "Projects of Large Infrastructure for Research, Development, and Innovations" (LM2010005), is greatly appreciated.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze dne .....

Karel Horák

Abstract

Graph databases are establishing as a respectable alternative to the relational data stores. In this thesis theoretical aspects of the problem solved by graph databases are discussed and an algorithm for evaluating graph queries using the concept of tree decomposition is proposed and thoroughly discussed. This algorithm is based on optimality results on graph homomorphism and is inspired by Yannakakis' algorithm for acyclic database schemes. Our algorithm is then evaluated by a series of experiments.

Keywords

graph databases, graph theory, graph homomorphism, tree decomposition, complexity theory, database theory

Abstrakt

Grafové databáze se těší vzrůstající oblibě a nezřídka se stávají alternativou k datovým úložištím založeným na relační technologii. Zaměříme se na rozbor teoretických problémů řešených grafovými databázemi a navrhneme algoritmus pro efektivní vyhodnocování grafových dotazů za použití konceptu stromové dekompozice. Tento algoritmus, založený na teoretických výsledcích v oblasti homomorfismu grafů, je inspirovaný Yannakakisovým algoritmem pro acyklická databázová schémata. Efektivita navrhovaného algoritmu je otestována v řadě experimentů.

Klíčová slova

grafové databáze, teorie grafů, homomorfismus grafů, stromová dekompozice, teorie složitosti, teorie databází

Contents

CHAPTER 1

Introduction

In the information age data management has become an important discipline in computer science. It is a tedious task to handle information without any support — specialized systems for storing, retrieval and analysis of data were therefore asked for.

One of the oldest and still dominant systems for data manipulation are based on relational database theory. There are many commercially successful RDBMS (relational database management systems) that have been used for decades already — like PostgreSQL[20] having its roots in the Ingres Database (a former UC Berkeley project started in 1970s).

The key concept in RDBMS' is the concept of relation. For simplicity, relations can be seen as tables where each column has its header. An example of a relational database can be seen in Figure 1.1. Individual objects in the world are represented as rows of the tables and their interactions are captured by sharing same values in matching columns (e.g. we know that Donald Griffin, aged 23, is a student of Czech Technical University in Prague).

Name	**Surname**	**Age**	**Uni**
Tim	Rodgers	22	2
Lucy	Kelly	21	2
Donald	Griffin	23	1
John	Williams	21	2
Julie	Ross	22	1

(a) Students relation

Uni	**University name**
1	Czech Technical University in Prague
2	University of California, Berkeley

(b) University relation

Figure 1.1: Sample relational database

The popularity of relational databases is based on the existence of more or less standardized language, which allows designers to define the data model (data definition language, DDL) and users to access data in convenient way (data manipulation language, DML). This language is called SQL — Structured Query Language. In Figure 1.2 a simple SQL query

```
SELECT * FROM Students
JOIN University ON Students.Uni = University.Uni
```

Name	Surname	Age	Uni	University name
Tim	Rodgers	22	2	University of California, Berkeley
Lucy	Kelly	21	2	University of California, Berkeley
Donald	Griffin	23	1	Czech Technical University in Prague
John	Williams	21	2	University of California, Berkeley
Julie	Ross	22	1	Czech Technical University in Prague

Figure 1.2: Sample SQL query and its result

and its result are shown (database from Figure 1.1 is considered). Apart from the ease of use, the decades long history of relational databases makes them a trusted data store.

Many applications from recent years happened to feel the limits that the relational technology imposes. The need for alternative data stores gave rise to the so called NoSQL movement (often thought of as Not only SQL). In comparison with the relational databases, there is a huge diversity amongst NoSQL databases — both in the data model and language used to interact with them. There are multiple reasons why one may consider adopting some of the database management systems belonging under the NoSQL movement:

- **Scalability.** Relational databases are typically run on a single machine (this allows to ensure ACID[1] properties). Many NoSQL databases sacrifice consistency in order to allow the database to be distributed across multiple machines. This may lower costs for hardware and software infrastructure and may allow availability and partition tolerance properties from the CAP theorem.
- **Flexibility.** Working with relational databases can be seen as a two phase procedure. Firstly an expert designs a schema (i.e. "table headers"), then users start inserting new data and issue queries. Changing the schema of a live database is a tedious task. NoSQL databases are often schemaless — allowing users to store whatever data whenever needed.
- **Performance.** Some NoSQL database systems are extremely lightweight (e.g. key–value stores). Such database systems does neither provide rich functionality nor guarantee properties like consistency — but they typically allow serving huge amounts of queries, often in parallel.

[1] Atomicity, Consistency, Isolation, Durability

	DBMS	Model	Score
1	Oracle	Relational	1439.72
2	MySQL	Relationa	1272.45
3	MS SQL	Relational	1177.48
4	MongoDB	Document	267.24
5	PostgreSQL	Relational	262.34
		...	
23	Neo4j	Graph	26.80
		...	

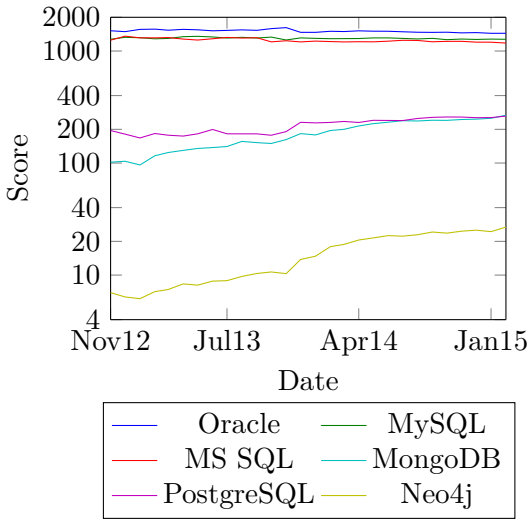


Figure 1.3: Popularity ranking of DBMS (DB-Engines.com, February 2015)

The adoption of NoSQL data stores is growing fast. According to DBMS ranking of DB–Engines.com[8] from February 2015 (excerpt in Figure 1.3), MongoDB has even beaten renowned relational database management system PostgreSQL in popularity.

Let us briefly review major branches of NoSQL databases — key–value stores, document databases and graph databases.

Key–value stores are the most lightweight databases one can think of. These systems serve as a large dictionary where user may store his data primitives (strings, numbers etc.) under a specific key and later retrieve them using the very same key. From the mathematical point of view, key–value stores may be seen as a function.

While key–value stores do not care about the structure of data users store, document databases ask users to store data in a structured way. The most common formats for expressing documents are XML and JSON. Document databases already provide a lot of functionality like querying documents by their contents (and not only by their identifiers) or updating just a portion of the document. But they are lacking a natural way of linking data from multiple documents together. A commonly used technique to overcome this issue is map–reduce. This however requires the user to deal with the data on a lower level.

We will focus on the branch of graph databases. These databases try to overcome the issue of document stores by adding a feature to express the linkage of individual vertices ("documents") in an explicit way. There is no common standard for graph databases — thus the only common thing is that data is captured in the form of a graph. In graph databases, objects are represented as vertices and their interactions are modelled by edges. A graph version of the database from Figure 1.1 is shown in Figure 1.4. The most popular graph database of these days is Neo4j[21] by Neo Technology (ranked 23 according to DB-Engines.com).

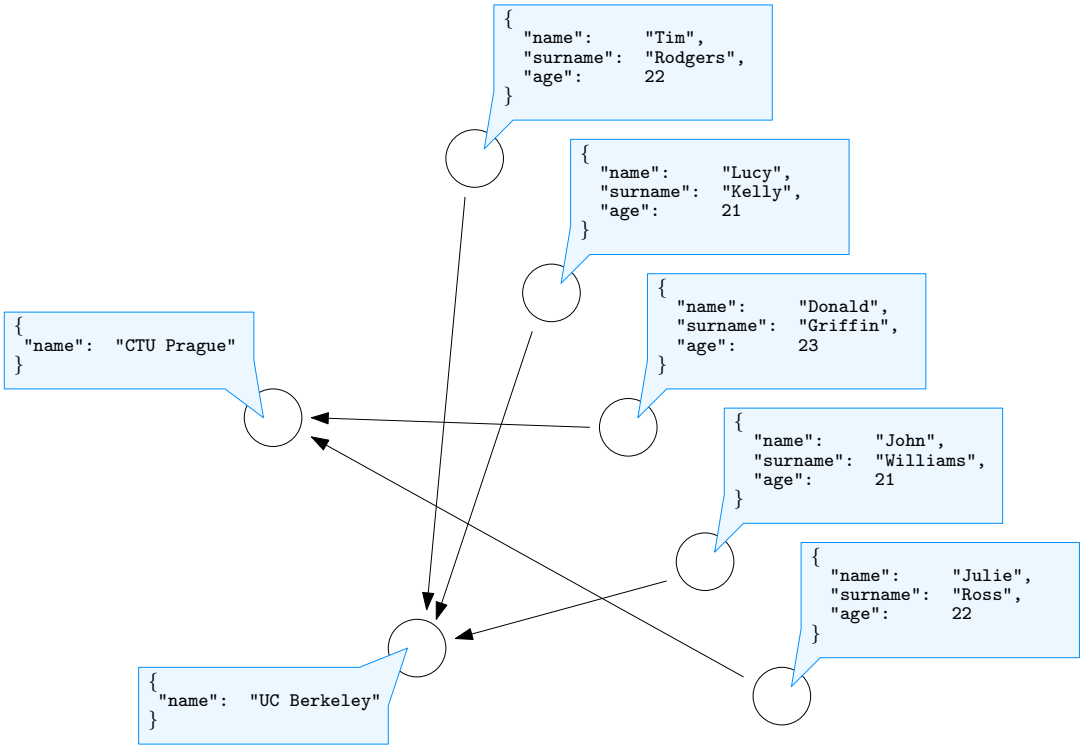


Figure 1.4: Sample graph database

In Chapter 2, graph databases are discussed from the mathematical point of view. Chapter starts by brief revision of basics of graph theory. The concept of graph homomorphisms is then introduced and related results are presented. The rest of the chapter is dedicated to the relation between graph homomorphisms and graph databases.

Even though graph databases significantly differ from the relational ones, it is worth being familiar with key concepts and results from the relational database theory — basics of this theory will be presented in Chapter 3.

These results are then used in Chapter 4 where an algorithm for evaluating graph queries is discussed. Chapter 5 is devoted to experimental evaluation of the algorithm from Chapter 4.

CHAPTER 2

Graphs, homomorphism and graph databases

The reader is assumed to be familiar with the basics of graph theory, nevertheless the first part of this chapter will be devoted to a quick revision and establishing notational conventions. In the end of this part, the concept of treewidth will be introduced.

Later in this chapter, the problem of graph homomorphism will be investigated and important theoretical results will be presented. This mainly includes complexity results of Nešetřil and Grohe.

In the end of this chapter, we will take a look at how graphs and the concept of graph homomorphism can be used to represent and query real world datasets. The problem that is solved by graph databases will be formally stated.

Graph is represented as a tuple $G = (V, E)$, where V denotes the set of *vertices* ("objects") and E denotes the set of *edges* ("interactions"). Edges can be either *undirected* (denoted $\{u, v\}$) in case of undirected graphs, or *directed* (denoted (u, v)) in case of directed graphs. Undirected graphs will be thought of unless stated otherwise. An undirected graph without loops is said to be *simple* if for every two vertices there is at most one edge connecting them. Graphs where multiple edges are allowed to connect same two vertices are called *multigraphs*.

When talking about graphs, declaration of sets V and E will often be omitted. Instead the set of vertices of graph G will be referred to as $V(G)$ and the set of edges as $E(G)$. We will be dealing only with finite graphs, i.e. those where both $V(G)$ and $E(G)$ are finite sets.

It is often convenient to visualize a graph using a diagram. In such a case, vertices are represented as points and edges as lines. A line connecting two points is present in the diagram if corresponding vertices are connected by an edge. Figure 2.1 exhibits such a diagram for a hypothetic friendship network (connecting two people by an edge means that they are friends). Notice that the actual positions of the points do not matter — the

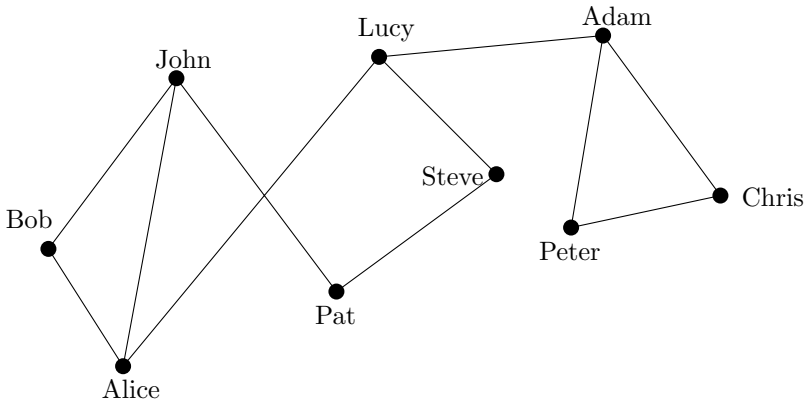


Figure 2.1: Friendship network

diagram captures only the interactions and not the positions of the vertices (there is no such information contained in the graph).

Let e be an edge connecting vertices u and v. Vertices u and v are called *endpoints* of e. Edge e is said to be *incident* to these vertices. If two vertices u, v are connected by an edge, then v is called to be *adjacent to* u (and vice versa).

Graph G' is a *subgraph* of G if $V(G')$ and $E(G')$ are subsets of $V(G)$ and $E(G)$. An *induced subgraph* of G by vertex set S (denoted $G[S]$) is a subgraph of G such that $V(G[S]) = S$ and all edges of G connecting vertices in S are preserved.

Sequence $v_0e_0v_1 \cdots v_{k-1}e_{k-1}v_k$ is a *path* in G if no two distinct v_i are the same and for every i, edge e_i connects vertices v_i and v_{i+1}. Graph G is said to be *connected* if for every two vertices u and v of G, there exists an undirected path from u to v. Vertex set S is said to be connected in G if $G[S]$ is connected.

A *connected component* of graph G is a maximal subgraph of G that is connected. A connected graph has exactly one connected component.

An *union* of graphs $G_1 \cup G_2 \cup \cdots \cup G_k$ is a graph G' such that $V(G) = \bigcup_i V(G_i)$ and $E(G) = \bigcup_i E(G_i)$.

Undirected graphs can be generalized to *hypergraphs*. Edges of hypergraphs are not restricted to connect exactly two vertices — an edge in a hypergraph can connect arbitrary number of vertices except for no vertices. To emphasize the difference between regular graphs and hypergraphs, edges of hypergraphs are often called *hyperedges*.

2.1 Tree decomposition

Treewidth is an important graph theoretical concept playing an important role in derivation of many *fixed parameter tractable* algorithms for otherwise $\mathcal{NP}$–hard problems[13]. This concept (alongside with *tree decomposition*) plays an important role in this thesis as well.

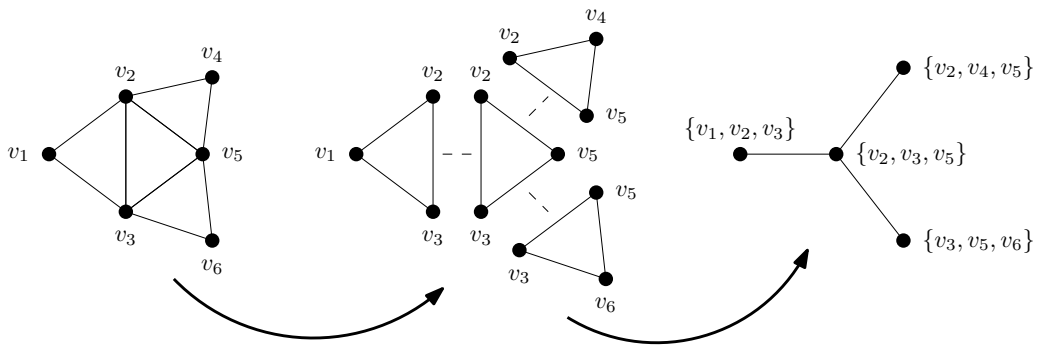


Figure 2.2: Graph and its tree decomposition

Definition 2.1 (Treewidth and tree decomposition)**.** Let G be a graph. Pair (T, χ), where T is a tree and $\chi : V(T) \to 2^{V(G)}$ is a mapping labeling each node of T by a subset of vertices of G, is called *tree decomposition* if following properties hold:

1. Every vertex $v \in V(G)$ is contained in $\chi(n)$ of some node n of T.
2. For every edge e of G connecting vertices u and v, there is a node n of T such that $\{u, v\} \subseteq \chi(n)$.
3. For every vertex v of G, the set of nodes n containing v in $\chi(n)$ is connected in T.

The *width* of (T, χ) is the maximum from cardinalities of sets $\chi(n)$ decreased by one (i.e. $\max_{n \in V(T)} |\chi(n)| - 1$). The minimal width over all tree decompositions of G is then the *treewidth* of G (denoted by $\mathbf{tw}(G)$). Sets $\chi(n)$ are called *bags*.

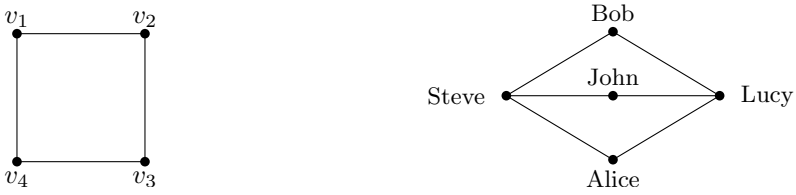
Remark. In order to distinguish between vertices of G and vertices of its tree decomposition, vertices of T will be called *nodes*.

Remark. In the remainder of the text, by tree decomposition of G the optimal tree decomposition of G (in terms of its width) will be thought of.

Figure 2.2 shows a graph and its tree decomposition with minimum width (width of this tree decomposition is 2).

In general case, computing optimal tree decomposition (with respect to the treewidth) is a $\mathcal{NP}$–hard problem[13]. However for graphs of small treewidth, efficient algorithms were proposed.

In 1991, Matoušek and Thomas[18] showed that an optimal tree decomposition of graphs of treewidth at most 3 can be computed in linear time. In 1996, Bodlaender[5] gave a linear time algorithm that for a constant k (fixed previously) either determines that treewidth of a graph is higher than k or constructs a tree decomposition with treewidth at most k.



(a) Query graph G **(b)** Graph database H

Figure 2.3: Sample graph query and database

2.2 Graph homomorphism

Graph homomorphism problem deals with finding a mapping from vertices of one graph to vertices of another one, such that adjacent vertices in the first one are adjacent in the second one as well. Let us first define the homomorphism mapping formally.

Definition 2.2 (Graph homomorphism)**.** Let G, H be graphs. Mapping $h : V(G) \to V(H)$ is called *homomorphism* if for every edge e of graph G, there is an edge of H connecting images of the endpoints of e (i.e. if $e = \{u, v\} \in E(G)$, then $\{h(u), h(v)\} \in E(H)$). Graph G from this definition is referred to as the *guest graph* (or the *query graph*). Graph H is a *host graph*. The set of all homomorphism mappings from G to H will be denoted by $Hom(G, H)$.

Let us illustrate the concept of graph homomorphism from the graph database perspective. In a typical graph database query, user provides a pattern that prescribes how vertices should be connected and asks for (real world) objects interacting in that way. This gives us a straightforward way of interpreting such queries in terms of graph homomorphisms — the query corresponds to graph G, whereas the database is graph H. Figure 2.3 shows an example of both query graph and graph database of friends.

Let us restrict our attention to homomorphisms such that $h(v_1) =$ John. The query can be seen as an attempt to build a simple system for recommendation of new friends — assuming that if someone is a friend of two friends of mine, he might be a friend of myself as well.

Some of the homomorphisms will indeed work in exactly the expected way. We will obtain four homomorphisms shown in Figure 2.4 (notice that in fact there are just two interesting results — the other two are just mirrored).

In this case, $h(v_3)$ is the person to recommend. Vertex v_3 has two adjacent vertices v_2 and v_4 — these correspond to those two common friends of John and $h(v_3)$.

Not all graph homomorphisms comply with our original intention — homomorphisms allow that multiple vertices of G are mapped on a single vertex. We will see that in such a case, this might lead to unexpected results. Figure 2.5 shows some of such results. In the

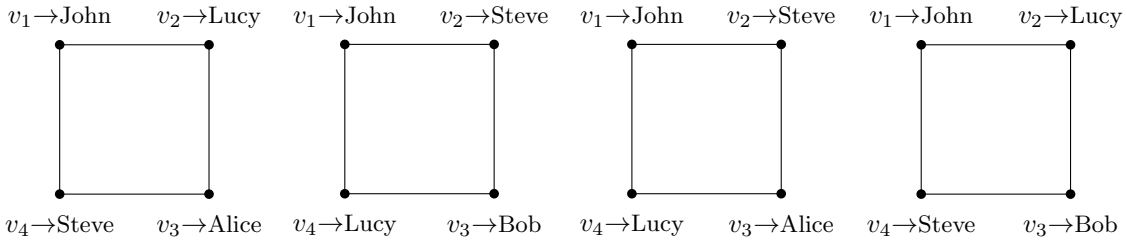


Figure 2.4: Four injective homomomorphisms

first of them, we would recommend John to become friend with Bob, but there seems to be just one common friend — Lucy. The remaining two figures would even propose that John should become a friend with himself.

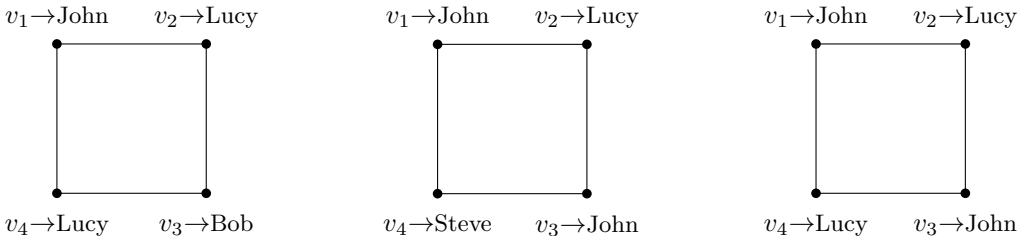


Figure 2.5: Non-injective homomorphisms

The graph homomorphism problem exists in multiple variants — the most typical one being the decision version. The unrestricted homomorphism problem is $\mathcal{NP}$–complete[11], thus it makes sense to study restricted versions of the problem. Let us first discuss restricting the input. Let $\mathcal{C}$, $\mathcal{D}$ be classes of graphs. We are asked a question: *Given two graphs $G \in \mathcal{C}$, $H \in \mathcal{D}$, does homomorphism from G to H exist?* In case of existence, we will write $G \to H$. The decision problem for classes $\mathcal{C}$ and $\mathcal{D}$ will be denoted by $\textsc{Hom}(\mathcal{C}, \mathcal{D})$. An algorithm decides $\textsc{Hom}(\mathcal{C}, \mathcal{D})$ if for every $G \in \mathcal{C}$, $H \in \mathcal{D}$ it correctly distinguishes instances where $G \to H$ from those where $G \not\to H$. In case that the input does not comply with classes $\mathcal{C}$ and $\mathcal{D}$, the behaviour of the algorithm may be arbitrary.

In 1990, Hell and Nešetřil[15] (as restated by Grohe[14]) showed that $\textsc{Hom}(—, \mathcal{D})$[1] is decidable in polynomial time if class $\mathcal{D}$ (of simple undirected graphs) contains only bipartite graphs. Otherwise the problem is $\mathcal{NP}$–complete. In the context of graph databases, class $\mathcal{D}$ contains possible states of graph store. Allowing user to store only bipartite graphs would be highly problematic and such a database would most likely be of no use.

On the other hand, class $\mathcal{C}$ corresponds to the queries an user can pose to the database. The problem seen from the other side, i.e. $\textsc{Hom}(\mathcal{C}, —)$, is therefore more relevant for us as these queries are typically rather small and simple. It is known that if class $\mathcal{C}$ has bounded

[1] Class of all simple undirected graphs is denoted by —

treewidth, $\textsc{Hom}(\mathcal{C}, —)$ is polynomial time solvable. The reasoning behind this claim will be shown in Chapter 4 where an algorithm that will later be used for empirical comparison with Neo4j database is derived.

It turns out that tractability of $\textsc{Hom}(\mathcal{C}, —)$ with class $\mathcal{C}$ having bounded treewidth is not the strongest result one can hope for. Our attention may focus on substituting an original problem instance (G, H) $(G \in \mathcal{C})$ with another one (G', H), such that $G \to H$ if and only if $G' \to H$ (and deciding about $G' \to H$ is easier than the original problem). This holds if graphs G and G' are *homomorphically equivalent* — i.e. simultaneously $G \to G'$ and $G' \to G$.

This is a reasonable idea for the decision variant of the problem, however it is hard to use this knowledge to construct the set of all homomorphism mappings $Hom(G, H)$. This problem will be demonstrated on the instance from Figure 2.6. In this figure graphs G, G' and H are shown (graphs G and G' are homomorphically equivalent). Homomorphism mappings from G to G' and from G' to H are shown in the figure — by composition of these mappings a homomorphism from G to H is obtained. This is fully satisfactory for deciding $G \to H$, but insufficient for constructing $Hom(G, H)$. One can check that sets $Hom(G, G')$ and $Hom(G', H)$ contain a single homomorphism mapping — therefore only one composed homomorphism from G to H can be obtained. It is clear that multiple homomorphisms from G to H can however be constructed.

In 2002, Dalmau et al.[7] showed that if class $\mathcal{C}$ has *bounded treewidth modulo homomorphic equivalence* (i.e. for fixed k, every $G \in \mathcal{C}$ has some homomorphically equivalent graph G' with $\mathbf{tw}(G') \leq k$ — this class will be denoted by $\mathcal{H}(\mathcal{T}^k)$), $\textsc{Hom}(\mathcal{C}, —)$ is decidable in polynomial time. To illustrate the importance of this class, consider class $\mathcal{C}$ of bipartite graphs. This class has unbounded treewidth, but it has bounded treewidth modulo homomorphic equivalence. Every bipartite graph with at least one edge is homomorphically equivalent with a graph with two vertices connected by an edge (treewidth of such a graph is 1).

On the other hand, same authors showed that for some fixed treewidth $k \geq 2$, testing membership in $\mathcal{H}(\mathcal{T}^k)$ is $\mathcal{NP}$–complete. This means that a polynomial time algorithm for deciding problem $\textsc{Hom}(\mathcal{H}(\mathcal{T}^k), —)$ exists, but if it is not guaranteed that $G \in \mathcal{H}(\mathcal{T}^k)$ (which is usually not known in advance), it is $\mathcal{NP}$–complete to verify correctness of the answer.

Grohe[14] later proved that the result of Dalmau is optimal in the sense, that for classes $\mathcal{C}$ of unbounded treewidth modulo homomorphic equivalence, problem $\textsc{Hom}(\mathcal{C}, —)$ is not in polynomial time.

Apart from restricting classes $\mathcal{C}$ and $\mathcal{D}$, we may restrict acceptable homomorphism mappings. The surjective variant of the homomorphism problem caught some attention recently. Surjectivity requires that for every vertex v of H, there is some vertex of G that is mapped on v. This makes this version of the problem be of little relevance for our goal — intuitively the database is supposed to hold large amount of data (and contain a lot of

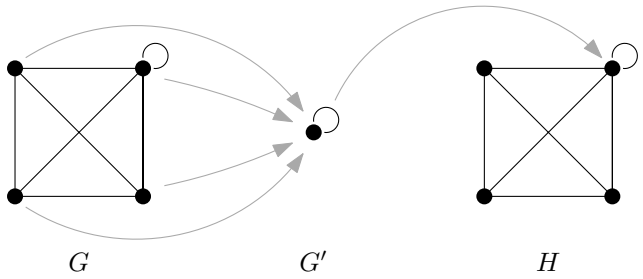


Figure 2.6: Deciding $G \to H$ using simpler homomorphically equivalent graph G'

vertices), hence the query cannot be required to cover all the objects in the database. For an overview of the surjective homomorphism, reader may consult paper of Golovach et al.[12].

On the contrary, injectivity of the homomorphism may be a reasonable requirement in the domain of graph databases — but it will be shown later in this chapter that this reasoning may easily become rather counterintuitive. The injective homomorphism problem can be seen as a problem of finding subgraph of H that is isomorphic to G — that is why this problem is commonly called *subgraph isomorphism* problem. This problem is deeply studied, e.g. in the paper of Marx and Pilipczuk[17].

The $\mathcal{NP}$–hardness of the general subgraph isomorphism problem can be easily seen as several well known $\mathcal{NP}$–complete problems, such as finding Hamiltonian cycle or k–clique, are its special cases — the case of Hamiltonian cycle problem even suggests that bounding the treewidth of G does not help. Every cycle has a treewidth 2 — if a polynomial time algorithm for deciding inj-Hom$(\mathcal{C}, —)$ for class $\mathcal{C}$ of bounded treewidth existed, the Hamiltonian cycle problem would have been in polynomial time (which would imply $\mathcal{P} = \mathcal{NP}$).

Most of the positive results stated in the aforementioned paper are of little use for our goal. Many of these algorithms are parametrized in terms of host graph (i.e. database, which could end up in unpredictable performance) or even restrict class $\mathcal{D}$ (which was already mentioned to be inacceptable). The most positive result from our perspective is therefore due to Alon, Yuster and Zwick (as stated by Marx and Pilipczuk):

Theorem 2.1. *Subgraph isomorphism problem can be solved in time* $2^{\mathcal{O}(|V(G)|)}n^{\mathcal{O}(\mathbf{tw}(G))}$.

Let us illustrate the difference between unrestricted and injective homomorphism on examples. Figure 2.7 shows graphs G and H (such that $G \to H$) where injective homomorphism does not exist. On the other hand, injective homomorphism exists in Figure 2.8.

There is a closely related notion to the subgraph isomorphism. The subgraph of H mentioned above may be required to be induced subgraph of H — this gives rise to the *induced subgraph isomorphism* problem. In terms of homomorphism mapping, the induced subgraph isomorphism requires that an edge is present in G if and only if images of its

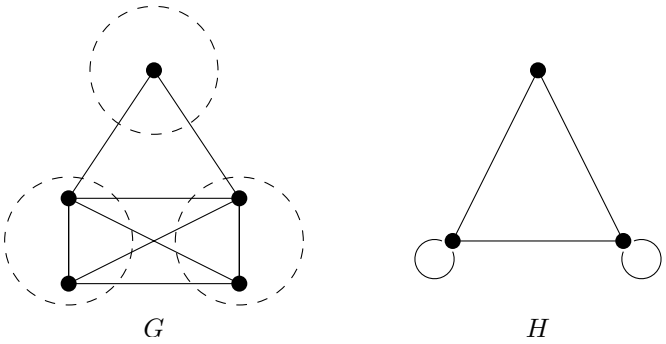


Figure 2.7: Injective homomorphism from G to H does not exist

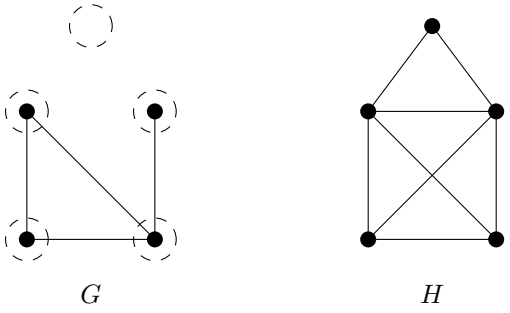


Figure 2.8: Injective homomorphism from G to H exists

endpoints are connected in H. This distinguishes it from the homomorphisms where the requirement that if two vertices of G are disconnected, their images must be disconnected in H as well is omitted.

The question whether there exists a homomorphism from G to H might not be the only question we are interested to answer — in fact in the context of graph databases, this question is probably the least significant one.

The most typical query databases (of whatever kind) are asked to answer is the enumerative one. In such a case, user is interested in computing all valid answers to the query he posed. In the case of graph homomorphism this means that the database is asked to compute all homomorphisms from G to H (or at least some well defined portion of $Hom(G, H)$ — in case of using some analogy of `LIMIT` clause from the family of SQL languages).

Apart from enumerating the set of all homomorphisms, user is often interested in knowing how many distinct homomorphism mappings from G to H exist (i.e. what is the cardinality of $Hom(G, H)$). This problem is called the counting problem.

To complete the list of homomorphism related problems, an algorithm that either decides that no homomorphism from G to H exists or constructs one such homomorphism may be asked for — this problem is closely related to the constraint satisfaction.

Following propositions capture two useful properties of homomorphisms — homomorphisms can be restricted and the result is still a homomorphism, while independent homomorphisms can be assembled together.

Proposition 2.1. *Let G, H be graphs and h be a homomorphism mapping from G to H. For an arbitrary subgraph G' of G, mapping $h|_{V(G')}$ (where $h|_{V(G')}$ denotes restriction of h to the vertex set of G') is homomorphism from G' to H.*

Proof. Mapping $h|_{V(G')}$ is clearly a mapping from $V(G')$ to $V(H)$, thus only the adjacency preservation condition has to be checked. As graph G' is a subgraph of G, it contains only subset of edges of G (i.e. $E(G') \subseteq E(G)$). By our assumption that h is a homomorphism mapping, we know that for every edge of G, images of its endpoints are connected in H. This must also hold for a subset of $E(G)$, thus $h|_{V(G')}$ is a homomorphism from G' to H. □

Proposition 2.2. *Let G and H be graphs and $G_1, \cdots, G_k$ be all connected components of G. Let $h_1, \cdots, h_k$ be mappings such that h_i is a homomorphism from G_i to H. Then mapping h defined as follows is a homomorphism from G to H:*

$$h(v) = \begin{cases} h_1(v) & \text{if } v \in V(G_1) \\ \vdots & \\ h_k(v) & \text{if } v \in V(G_k) \end{cases}$$

Proof. The union of connected components $G_1, \cdots, G_k$ is equal to the graph G, therefore h is a mapping from $V(G)$ to $V(H)$. Mapping h is a homomorphism from G to H as no edge of G has its endpoints in two different connected components G_i. □

Proposition 2.2 allows us to focus on connected query graphs only. In case of disconnected graph G, we may identify its connected components first and solve the homomorphism problem for each one of them separately.

2.3 Graph databases

Graph databases exploit the concept of the graph to store data. In this work, the most successful graph database at the moment will be discussed — Neo4j. The expressivity of this database system is comparable with the expressivity of relational databases. This required to introduce few additional features to the graphs that were not discussed previously.

In Figure 1.1 a very simple example of relational database was presented. This database not only captures interactions between students and universities. Both students and universities are attributed additional information — e.g. the age of the students. Neo4j solves this by assigning metadata to the vertices. These metadata are captured in Javascript Object Notation (JSON). Figure 2.9 contains an example of data representation in JSON format. Part of the JSON grammar is shown in Figure 2.10.

The actual membership of a row to a table is an important information itself — it provides an information about the type of data. Neo4j allows users to assign labels to vertices, e.g. user can assign a vertex label :STUDENT if that vertex contains data of some student. The same options as for vertices are provided for edges in Neo4j. Like that users can specify that a student is indeed a student of an university (and not an employee), by assigning the edge a label of :IS_STUDENT.

So far we have been dealing with symmetric friendship relation. Some of the interactions are however clearly asymetric (e.g. role of manager and his subordinate). Therefore it makes sense to deal with directed graphs.

In traditional graph theory, graphs contain either directed or undirected edges (exclusively). If an undirected edge should be present in a directed graph, this edge is represented as a pair of oppositely directed edges. This can be done in graph databases as well — but one has to be careful. The undirected edge in a graph database is a single object with its labels and metadata. Thus either every update operation has to keep both edges in sync, or directed and undirected edges should be stored separately. For the sake of simplicity we will adopt the former approach.

It was mentioned that both injective and unrestricted versions of the homomorphism problem are reasonable choices for evaluating graph database queries. The evaluation of the queries in Neo4j is based on the unrestricted variant. There are several reasons for this choice.

Firstly, there are historical reasons for this choice. SQL queries in relational database management systems are evaluated as non–injective as well.

Secondly, we have seen that evaluating injective homomorphism is exponential in the size of the query graph according to Theorem 2.1. A graph database based on injective homomorphisms could not provide any reasonable time complexity guarantees.

Last but not least, the injective homomorphisms might easily get counterintuitive. An intuitive way of interpreting query in Figure 2.11 would be *Give me all friends of Bob's friends*. An injective homomorphism would forbid assigning Bob to vertex v_3 — which is somewhat unwanted behaviour, as in real life, Bob is a friend of every of his friends.

In this thesis, we will consider only the structural part of the graph database — i.e. vertices, edges and their labels. Let us state the homomorphism problem for graph databases formally.

Definition 2.3 (Labeled graph)**.** Let Λ be the universe of labels. *Labeled graph* is a pair (G, λ) where G is a directed multigraph and $\lambda : V(G) \cup E(G) \to 2^{\Lambda}$ is a *labeling function*.

Remark. Bold font will be used for distinguishing labeled graphs from regular ones (e.g. $\mathbf{G}$). Graph concepts and notation are extended to labeled graphs in a natural way (e.g. $V(\mathbf{G})$ denotes vertex set of $\mathbf{G}$).

Disregarding metadata, both the query graph and host graph are labeled graphs in the context of graph databases.

Definition 2.4 (Homomorphism for graph databases)**.** Let $\mathbf{G} = (G, \lambda_{\mathbf{G}})$ be a query graph and $\mathbf{H} = (H, \lambda_{\mathbf{H}})$ be a host graph. Mapping $h : V(\mathbf{G}) \to V(\mathbf{H})$ is called a *homomorphism* if it satisfies following properties:

- For every vertex $v \in V(\mathbf{G})$, $\lambda_{\mathbf{G}}(v) \subseteq \lambda_{\mathbf{H}}(h(v))$.
- For every edge $e = (u, v) \in E(\mathbf{G})$ connecting vertex u with v, there is an edge $e' = (h(u), h(v)) \in E(\mathbf{H})$ connecting vertex $h(u)$ with $h(v)$ such that $\lambda_{\mathbf{G}}(e) \subseteq \lambda_{\mathbf{H}}(e')$.

The homomorphism in the context of graph databases is similar to the regular graph homomorphism, except for the fact that also the labels have to be preserved.

Let us illustrate the homomorphism for graph databases on an example. Figure 2.12 shows graphs $\mathbf{G}$ and $\mathbf{H}$ such that there is only a single homomorphism mapping — assigning *John* to v_1 and *The Pickup* to v_2. Vertex v_1 cannot be mapped to *Lucy* as there is no car she owns.

The reason for the growing popularity of the Neo4j database might be caused by the existence of convenient language for posing queries to the database engine. The language used in Neo4j is called Cypher. We won't cover this rich language in detail — we will just show an example how the query from Figure 2.12 can be rewritten in Cypher. This Cypher formulation is shown in Figure 2.13.

Number of homomorphisms from $\mathbf{G}$ to $\mathbf{H}$ may grow exponentially in the size of $\mathbf{G}$ — one cannot therefore expect to construct an algorithm to enumerate $Hom(\mathbf{G}, \mathbf{H})$ (for varying $\mathbf{G}$) in polynomial time under any parametrization. In practical applications, the number of results is typically much lower. It makes therefore sense to analyze the complexity of algorithms in the size of the input and the output.

One of the simplest algorithm for finding homomorphisms is the backtracking algorithm. The key concept in this algorithm is a partial homomorphism. Let $\mathbf{G}'$ be an induced subgraph

of $\mathbf{G}$ (induced by vertex set V'). Partial homomorphism p from $\mathbf{G}$ to $\mathbf{H}$ is a homomorphism from $\mathbf{G}'$ to $\mathbf{H}$. In every step, this algorithm attempts to extend set V' by one vertex (such that the resulting mapping is once again a partial homomorphism). If it fails to do so, it has to alter current partial homomorphism — it backtracks. A recursive version of this algorithm is shown in Figure 2.14.

This algorithm may seem naïve, but in fact similar algorithms are often used. This algorithm traverses the space of partial solutions in depth first search manner — this results in its memory efficiency.

The main drawback of the backtracking algorithm is its time complexity. Even in the case of simple queries (with low fixed treewidth), the worst case time complexity of the algorithm is exponential in the size of the input regardless of the size of the output. Figure 2.15 shows an example of graphs $\mathbf{G}$ and $\mathbf{H}$ that cause problems to the backtracking algorithm. Checking that directed cycle cannot exist in a directed acyclic graph would solve this particular instance, but the overall problem persists. Instances similar to the one shown in Figure 2.15 will be used in Chapter 5 to show that Neo4j runtime may become exponential in the size of $\mathbf{G}$ (i.e. the length of the cycle in $\mathbf{G}$).

```
{
  "name":        "John",
  "surname":     "Doe",
  "age":         24,

  "occupation": [
    "computer vision",
    "computational graphics"
  ],
  "address": {
    "street": "99 Cambridge Road",
    "city":   "North Cove"
  }
}
```

Figure 2.9: Sample JSON content

$$
\begin{array}{rcl}
json & ::= & object \mid array \mid string \mid number \mid boolean \\
object & ::= & \{\} \mid \{field\_list\} \\
field\_list & ::= & field \mid field,field\_list \\
field & ::= & identifier{:}json \\
identifier & ::= & string \\
array & ::= & [] \mid [values] \\
values & ::= & json \mid json,values \\
boolean & ::= & \text{true} \mid \text{false}
\end{array}
$$

Figure 2.10: Part of JSON grammar

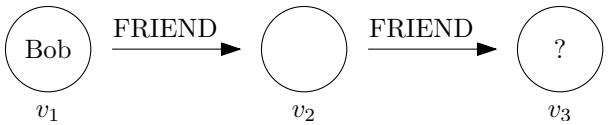
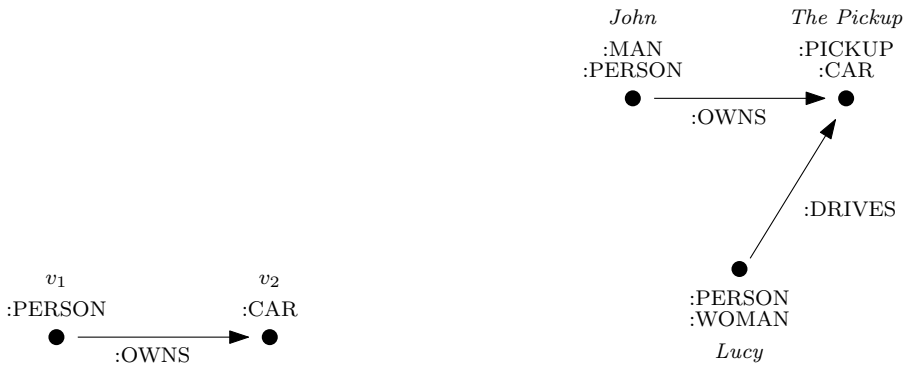


Figure 2.11: Friends–of–Friends query



(a) Query graph **G** **(b)** Graph database **H**

Figure 2.12: Graph database homomorphism example

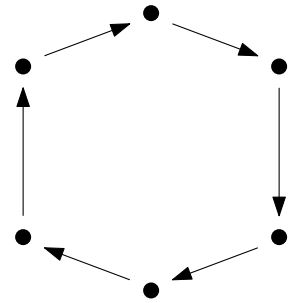
```
MATCH (v1:PERSON) -[r:OWNS]-> (v2:CAR)
RETURN v1, r, v2
```

Figure 2.13: Sample Cypher query

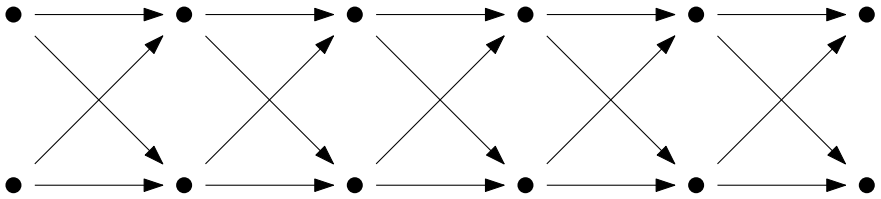
Require: query graph **G**, host graph **H**

procedure COMPUTE(V', p)
  if $V' = V(\mathbf{G})$ **then**
    report homomorphism p
  else
    $v \leftarrow$ some element of $V(\mathbf{G}) \setminus V'$
    for all vertices $v' \in V(\mathbf{H})$ such that $p \cup \{v \to v'\}$ is a partial homomorphism **do**
      COMPUTE($V' \cup \{v\}$, $p \cup \{v \to v'\}$)
    end for
  end if
end procedure

Figure 2.14: Recursive version of backtracking algorithm



(a) Query graph **G**



(b) Host graph **H**

Figure 2.15: Problematical instance for backtracking algorithm

CHAPTER 3

Relational database theory

In order to derive an efficient algorithm for evaluating queries in a graph database, reader is required to have some background on relational database theory. In Chapter 1, the main idea behind relational database management systems was shown from the practical point of view. In practice, users of RDBMS indeed think of relations as of tables (and they even call them tables), and the individual entries are called records (or rows). The theoretical framework of relational database theory is however more abstract.

In this chapter, foundations of relational database theory are presented. First part of the chapter is devoted to formal definitions of key terms and relational operators. In the rest of the chapter, the theory is seen from the computational point of view — the concept of acyclic database scheme is presented and Yannakakis' algorithm for acyclic database schemes is discussed.

3.1 Relations and databases

In Chapter 1, we were thinking of relations as of tables where each column has a header. Before a relation is defined formally, we have to make sure what the header means.

Definition 3.1 (Relation scheme)**.** A *relation scheme* is a finite set of *attributes* $\{A_1, \cdots, A_n\}$. Each of the attributes is assigned a *domain*. The domain of attribute A_i is denoted by $dom(A_i)$.

Remark. By convention, relation schemes will be denoted by capital R.

To illustrate this definition, let us consider the example from Figure 1.1. The relation scheme for student relation is the set $\{\text{Name}, \text{Surname}, \text{Age}, \text{Uni}\}$ with appropriately set

domains, e.g. dom(Surname) might be a set of all surnames and dom(Age) be a set of natural numbers (possibly including zero).

Definition 3.2 (Tuple)**.** Let $R = \{A_1, \cdots, A_n\}$ be a relation scheme and D be an union of domains of its attributes (i.e. $D = dom(A_1) \cup \cdots \cup dom(A_n)$). A *tuple over relation scheme* R is a mapping $t : R \to D$ such that for every attribute $A_i \in R$, $t(A_i) \in dom(A_i)$.

Intuitively, for every record in the table, we are interested in knowing its values in every column. These values are easily obtained from a tuple — by passing it an attribute. As the relation scheme restricts possible values in the columns, not every tuple is valid — it must match the domains of the attributes.

Definition 3.3 (Relation)**.** A *relation over the relation scheme* R is a set of tuples over relation scheme R.

Remark. If R is a relation scheme, a relation over R will be denoted by lowercase r.

It is indeed reasonable to view relations as tables, but one has to be aware of the differences. The definition of relation using sets and mappings does not explicitely define an order of columns and rows. In case of tables, we may be tempted to think that the order of columns and rows is fixed.

Furthermore, as relations are sets of tuples, a relation cannot contain two duplicate tuples. This differs from the common practice in majority of contemporary RDBMS, where the deduplication of tuples has to be done explicitely by the schema designer using the `UNIQUE` constraint.

In Figure 1.1, two relations shown are closely related — in order to get full information, data from both tables have to be combined.

Definition 3.4 (Database scheme)**.** A *database scheme* is a set of relation schemes.

Remark. Database schemes will be referred to by capital D.

Definition 3.5 (Database)**.** Let $D = \{R_1, \cdots, R_k\}$ be a database scheme. A *database* is a set of relations $\{r_1, \cdots, r_k\}$ such that r_i is a relation over relation scheme R_i.

Remark. Similarly as in the case of relation schemes and relations, lowercase d will be used for databases.

Relation scheme defines a dataless template for relations — and relations are instances containing data. The same applies for database scheme and database; database scheme prescribes a template for databases (all their relations), whereas the database fills in data in the form of relations.

3.2 Relational algebra

In the first chapter, a language for practical database manipulation was mentioned. Despite different vendors provide different dialects of SQL language, there is one thing in common — SQL language provide a lot of functionality and it is rather complex. For simpler analysis of relational system, a simpler and well defined language is necessary — the language of relational algebra.

Relations are defined as sets, therefore set operators like union ($\cup$), intersection ($\cap$) or set difference ($\setminus$) can be applied. One has to check that both relations involved are over the same relation scheme in order to obtain a result that is a relation itself. We assume that the reader is familiar with these basic set operations. We will focus on operators that are specific for the relational database theory.

3.2.1 Project operator

Definition 3.6 (Project operator)**.** Let r be a relation over relation scheme R and $A \subseteq R$ be a subset of R's attributes. By *projection of r on A*, following relation over A is understood:

$$\pi_A(r) = \{\, t|_A \mid t \in r \,\},$$

where $\cdot|_A$ is a restriction of a mapping on A. The projection of r on A is denoted by $\pi_A(r)$.

Remark. For reasons of notational convenience, the project operator will often be applied on tuples. In such a case it is just an alias for the restriction of the mapping, i.e. $\pi_A(t) = t|_A$.

Intuitively, we can understand the project operator as an operator that deletes some columns from a relation. Figure 3.1 shows a projection of students table on attributes Age and Uni. Notice that one tuple got lost by applying project operator — tuples $(\text{Lucy}, \text{Kelly}, 21, 2)$ and $(\text{John}, \text{Williams}, 21, 2)$ are both mapped on tuple $(21, 2)$ by $\pi_{\{\text{Age,Uni}\}}$.

3.2.2 Join operator

In Chapter 1, the importance of combining information from multiple tables was shown on an example (in that case by means of SQL language). The join operator perform this

Name	Surname	Age	Uni
Tim	Rodgers	22	2
Lucy	Kelly	21	2
Donald	Griffin	23	1
John	Williams	21	2
Julie	Ross	22	1

(a) Relation r

Age	Uni
22	2
21	2
23	1
22	1

(b) Projection $\pi_{\{\text{Age,Uni}\}}(r)$

Figure 3.1: Project operator example

operation in the language of relational algebra. We will define this operator in a slightly atypical way — studying tuples first and then proceeding to whole relations.

Definition 3.7 (Join–compatible tuples)**.** Tuples t_1, t_2 over relation scheme R_1, R_2 are said to be *join–compatible* if $\pi_{R_1 \cap R_2}(t_1) = \pi_{R_1 \cap R_2}(t_2)$.

Set of tuples $\{t_1, \cdots, t_k\}$ is join–compatible if for every $i \neq j$ tuples t_i and t_j are join–compatible.

Simply speaking, two tuples are join–compatible if they do not assign different values to the same attribute. The join–compatibility allows tuples to be joined in the sense of the following definition:

Definition 3.8 (Join operator)**.** Let t_1, t_2 be join–compatible tuples over relation schemes R_1, R_2. *Join of* t_1 *with* t_2 is a tuple $t_1 \bowtie t_2$ over relation scheme $R_1 \cup R_2$, such that:

$$(t_1 \bowtie t_2)(a) = \begin{cases} t_1(a), & a \in R_1 \\ t_2(a), & a \in R_2 \setminus R_1 \end{cases}$$

Remark. If tuples t_1, t_2 are not join–compatible, the result of the join is undefined (possibly a failure).

The join operator is extended towards relation in the following way. Let r_1, r_2 be relations over relation schemes R_1, R_2. *Join of* r_1 *with* r_2 is a relation over $R_1 \cup R_2$ denoted $r_1 \bowtie r_2$ defined as follows:

$$r_1 \bowtie r_2 = \{\, t_1 \bowtie t_2 \mid \text{join–compatible tuples } t_1 \in r_1, t_2 \in r_2 \,\}$$

Remark. Join of two relations is a relation of all possible joins of tuples from these two relations.

Remark. Join operator is both commutative and associative.

Figure 3.2 shows join–compatible tuples for the database from Figure 1.1. These tuples are used to form a relation Students ⋈ Universities shown in Figure 3.3.

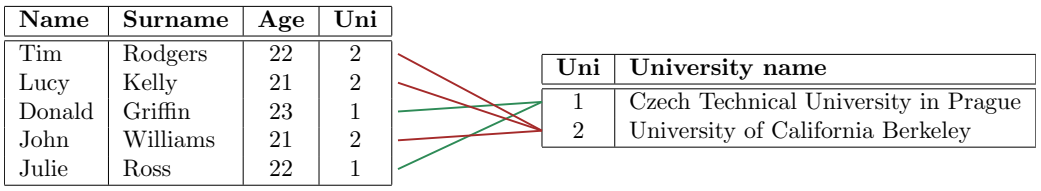


Figure 3.2: Join–compatible tuples (for relations Students and Universities)

Name	Surname	Age	Uni	University name
Tim	Rodgers	22	2	University of California, Berkeley
Lucy	Kelly	21	2	University of California, Berkeley
Donald	Griffin	23	1	Czech Technical University in Prague
John	Williams	21	2	University of California, Berkeley
Julie	Ross	22	1	Czech Technical University in Prague

Figure 3.3: Relation Students ⋈ Universities

3.3 Consistency and Joins

The join operator takes only join–compatible tuples into account. Tuples that are not join–compatible with any tuple from joined relation are not relevant for the computation of the join at all. Such tuples can be safely removed without changing the result of the join. We will go through several concepts that are built around this observation.

Definition 3.9 (Consistent tuple)**.** Tuple t_1 over relation scheme R_1 is *consistent with relation* r_2 over relation scheme R_2 if and only if there is a tuple $t_2 \in r_2$ such that tuples t_1 and t_2 are join–compatible.

The definition is extended in straightforward fashion to the relations, when the consistency means non–existence of inconsistent tuple.

Definition 3.10 (Consistent relations)**.** Relation r_1 *is consistent with relation* r_2 if and only if every tuple of r_1 is consistent with r_2. Relations r_1 and r_2 are *consistent* if simultaneously r_1 is consistent with r_2 and vice versa.

To grasp a better intuition about joins and consistency, we will show that our statement about irrelevancy of inconsistent tuples for joins is correct.

Proposition 3.1. *Let r_1, r_2 be relations over R_1, R_2 and t be an inconsistent tuple of r_1. Then the following hold:*

$$r_1 \bowtie r_2 = (r_1 \setminus \{t\}) \bowtie r_2$$

Remark. Join is a commutative operator, thus we can swap the roles of r_1 and r_2.

Proof. Join operator produces a single row for every join–compatible pair of tuples $t_1 \in r_1$, $t_2 \in r_2$. We will show that the set of join–compatible pairs was left unchanged by the removal of an inconsistent tuple t. It is trivial to see that no new join–compatible tuple could have been created. It remains to show that no join–compatible pair could have been removed by the removal of t. As t was an inconsistent tuple with relation r_2, there exists no tuple $t' \in r_2$ such that $\pi_{R_1 \cap R_2}(t) = \pi_{R_1 \cap R_2}(t')$ — which is the necessary condition for join–compatibility. □

As it is possible to omit inconsistent tuples from the relations before computing joins, we are naturally interested in computing a relation containing only consistent tuples.

Definition 3.11 (Semijoin)**.** Let r_1, r_2 be relations over relation schemes R_1, R_2. A *semijoin* of relation r_1 with r_2 is a relation $r_1 \ltimes r_2$ over relation scheme R_1 defined as follows:

$$r_1 \ltimes r_2 = \pi_{R_1}(r_1 \bowtie r_2)$$

We can see that only inconsistent tuples get eliminated by applying the semijoin. All consistent tuples participate in forming at least one tuple in $r_1 \bowtie r_2$ — thus they are preserved after applying π_{R_1}. On the other hand, for inconsistent tuples, no join–compatible pair of tuples exists — hence no tuple is generated by them in $r_1 \bowtie r_2$.

A	B
0	0
1	1

(a) Relation r_1

B	C
0	0
1	1

(b) Relation r_2

A	C
0	1
1	0

(c) Relation r_3

Figure 3.4: Globally inconsistent database (but pairwise consistent) [3]

3.4 Database consistency

The concept of consistency for two relations can be extended towards databases containing multiple relations. The straightforward extension of consistency of relations is the pairwise consistency of databases.

Definition 3.12 (Pairwise consistency)**.** Database $d = \{r_1, \cdots, r_k\}$ is *pairwise consistent* if any pair of relations of d is consistent.

In case of consistency for relations, every consistent tuple (let us say tuple t over relation scheme R) was participating on the join. Therefore there must have been tuple t' in the join such that $\pi_R(t') = t$. The information represented by t was not lost by applying the join and we could have "reconstruct" the original content of the relations by applying the project operator.

In case of pairwise consistency the reconstruction part no longer holds. In Figure 3.4 a pairwise consistent database is shown. The result of computing $r_1 \bowtie r_2 \bowtie r_3$ is however an empty relation (one can check that no three tuples t_1, t_2, t_3, $t_i \in r_i$, are join–compatible). The information contained in relations r_1, r_2 and r_3 got lost. A stronger consistency concept is therefore needed.

Definition 3.13 (Global consistency)**.** Database $d = \{r_1, \cdots, r_k\}$ over database scheme $R = \{R_1, \cdots, R_k\}$ is *globally consistent* if every relation of d can be obtained from the join by means of project operator:

$$r_i = \pi_{R_i}(r_1 \bowtie \cdots \bowtie r_k), \text{ for every } i \in [k]$$

Remark. Global consistency is usually defined by using existence of universal relation whose projections are relations $r_1, \cdots, r_k$. This is equivalent to our definition.

The definition of global consistency uses similar formula as a semijoin (except that the semijoin uses just join of two relations). In fact, if $r_i = \pi_{R_i}(r_1 \bowtie \cdots \bowtie r_k)$ for every $i \in [k]$, also $r_i = \pi_{R_i}(r_i \bowtie r_j)$ for every $i \neq j$. Thus for every pair of relations, application of semijoin

does nothing, and relations r_i, r_j are consistent. Hence global consistency guarantees pairwise consistency (but as was shown in Figure 3.4, converse is not true).

3.5 Acyclic database schemes

The most important class of database schemes is the class of acyclic database schemes. Efficient algorithms are often applicable for acyclic structures — and acyclic database schemes are of no exception. Many $\mathcal{NP}$–complete problems for general database schemes are polynomial time solvable in acyclic ones[2]. Beeri et al.[3] characterized acyclic database schemes by 9 equivalent statements. Some of them will be used later in the text and will be discussed thoroughly.

In the rest of this section we consider database scheme $D = \{R_1, \cdots, R_k\}$. By A we will denote the set of all attributes used in D (i.e. $A = R_1 \cup \cdots \cup R_k$).

Pairwise consistency implies global consistency.

We have shown in Figure 3.4 that for general database schemes we cannot expect that pairwise consistent database is also globally consistent. From the computational point of view, an important property of acyclic database schemes is that global consistency is guaranteed by pairwise consistency[3].

Every database over D has a full reducer.

A database over acyclic database scheme can be made globally consistent by applying a *semijoin program*. Such a program consists of a sequence of actions of the following form:

$$r_i \leftarrow r_i \ltimes r_j$$

We have shown that a relation r_i can be made consistent with r_j by applying semijoin $r_i \leftarrow r_i \ltimes r_j$. In databases over acyclic database schemes we can apply a semijoin several times in order to make the whole database pairwise consistent. Using previous characterization of acyclic database schemes, the global consistency is then obvious.

The construction of a semijoin program for turning database to a globally consistent state will be discussed later on in this section.

$(R_1, \cdots, R_k)$ has a join forest

The efficient computation of join of a database over acyclic database scheme is done using a structure called *join forest*. A join forest for a database scheme $D = \{R_1, \cdots, R_k\}$ is a forest where the relation schemes $R_1, \cdots, R_k$ are vertices. Every edge $\{R_i, R_j\}$ of the join forest is labeled by the set of shared attributes of the contributing relations — i.e. $R_i \cap R_j$.

Every two relation schemes R_i, R_j whose intersection $R_i \cap R_j$ is non–empty are connected by a path. Every edge on this path contains vertices from $R_i \cap R_j$ in its label.

The join forest structure is closely related to the tree decomposition. The path property of join forest is equivalent to the property of a tree decomposition stating that the node set containing a certain attribute is connected in the tree decomposition. The main difference between join forests and tree decompositions is that join forests can contain multiple connected components which is an unimportant technical detail in our scenario.

For the sake of completeness we will state the equivalence theorem in its entirety. (A, D) denotes a hypergraph where attributes of the database scheme form the vertices and relation schemes of D are the hyperedges.

Theorem 3.1 (1981, Beeri et al.[3]). *Let $D = \{R_1, \cdots, R_k\}$ be a database scheme and $A = R_1 \cup \cdots \cup R_k$ be a set of all attributes. Following statements about database D are equivalent:*

- *(A, D) is an acyclic hypergraph.*
- *(A, D) is a closed-acyclic hypergraph.*
- *(A, D) is a chordal hypergraph.*
- *The join dependency $\bowtie (R_1, \cdots, R_k)$ is equivalent to a set of multivalued dependencies.*
- *$(R_1, \cdots, R_k)$ has running intersection property.*
- *Following two operations reduce the list $(R_1, \cdots, R_k)$ to nothing, if applied repeatedly:*
  1. *delete attribute that is contained in a single relation scheme*
  2. *delete relation scheme R_i that is fully covered by another R_j (i.e. $R_i \subseteq R_j$, $i \neq j$)*
- *Pairwise consistency implies global consistency.*
- *Every database over D has a full reducer.*
- *$(R_1, \cdots, R_k)$ has a join forest.*

3.5.1 Computing full reduction

The importance of removing inconsistent tuples was already mentioned. A *full reduction* of a database d over database scheme D is a database d' over the same database scheme where no inconsistent tuples are present and joins of d and d' do not differ.

For the sake of simplicity, let us consider a join forest for D containing a single connected component — i.e. join tree. Let us root this tree and denote it by T. Let $R_1, \cdots, R_k$ be relation schemes of D in the order given by post–order traversal of T (i.e. relation scheme R_i comes after its children).

The computation of a full reduction is a two phase procedure. In the first phase, every r_i is made consistent with its children — T is traversed in bottom–up manner. This phase can be described by following semijoin program:

$$\text{for } i \text{ from 1 to } k \text{ and for every children } R_j \text{ of } R_i \text{ with respect to } T : r_i \leftarrow r_i \ltimes r_j$$

There are $k-1$ edges in T, hence this first phase consists of $k-1$ statements. The information propagates throughout the tree — therefore r_i is not consistent with its children only. r_i is consistent with all its successors with respect to T.

The second phase serves to propagate this information in the opposite direction — the tree is traversed top–down. The semijoin program for this phase may be seen as a mirrored version of the semijoin program for the first phase:

$$\text{for } i \text{ from } k \text{ downto 1 and for every children } R_j \text{ of } R_i \text{ with respect to } T : r_j \leftarrow r_j \ltimes r_i$$

This not only makes every relation be consistent with its predecessors. Due to the information flow in the algorithm, relations are made consistent also with relations in parallel branches of the join tree.

3.5.2 Evaluating project–join queries

Definition 3.14 (Project–join query)**.** Let d be a database over database scheme D and let S be a subset of attributes used in D (i.e. $S \subseteq \bigcup_{R \in D} R$). A *project–join query* over d is a statement of the following form:

$$\pi_S(\bowtie_{r \in d} r)$$

It was pointed out by Yannakakis [22] that project–join queries over database d can be evaluated in polynomial time in the size of input and output if the database scheme D is acyclic. Yannakakis' algorithm relies on two key properties of acyclic database schemes — the existence of full reducer and the existence of a join tree.

The first step in the evaluation of a project–join query $\pi_S(\bowtie_{r \in d} r)$ by means of Yannakakis' algorithm is to compute a full reduction d' of d as shown in Subsection 3.5.1. We know that joins of d and d' are equivalent, hence also $\pi_S(\bowtie_{r \in d} r) = \pi_S(\bowtie_{r' \in d'} r')$.

Let T be the join tree for D rooted in an arbitrary node. Yannakakis' algorithm traverses the tree in bottom–up manner. At every step one leaf relation is processed — its information

is merged into the parent's relation and the processed relation itself is discarded. This step is repeated until a single relation containing all the information (i.e. the result of the project–join query $\pi_S(\bowtie_{r \in d} r)$) remains.

Let R_i be the relation scheme of the leaf–node relation that is about to be removed and let r'_i be the relation currently associated to this node (these relations initially come from the reduced database d'). Let us denote the parent relation scheme of R_i by R_j — the relation associated to R_j is denoted by r'_j. The information from r'_i has to be integrated in the parent's relation r'_j. We can safely forget about the attributes that are not about to be present in the result (i.e. those that are not present in S) and at the same time they are irrelevant for the join with the parent's relation. Let S_i denote the set of attributes from S that are present in some of the relation schemes from the subtree of R_i. When performing join of r'_i and r'_j, attributes $Z_i = R_i \cap R_j$ are considered when deciding about join–compatible tuples. Thus the relation r'_i could be safely reduced to $\pi_{S_i \cup Z_i}(r'_i)$. The integration step is then straightforward — relation r'_j is replaced by the join $r'_j \bowtie \pi_{S_i \cup Z_i}(r'_i)$.

This algorithm works in polynomial time in the size of the input and the output as at every stage the size of relations r'_i is bounded by $|r_i| \cdot |\pi_S(\bowtie_{r \in d} r)|$ (as shown by Yannakakis[22]). It has to be mentioned that if the full reduction was not computed first, the size of relations r'_i could have become exponential in the size of the input and the output.

CHAPTER 4

Database theoretical view on homomorphisms

This chapter will present the connections between graph homomorphisms and relational database theory. Due to the maturity of database theory, identifying graph homomorphisms with database theoretical concepts is of great importance — allowing us to rely on algorithms and tools from this framework.

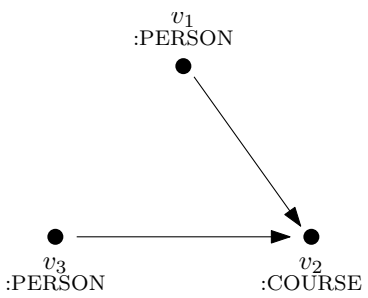
Firstly, let us investigate this connection without caring much about the complexity of the homomorphism enumeration. In Chapter 3, tuples were defined as mappings from the set of attributes to the universe. Homomorphism is a mapping as well. It is therefore straightforward to represent homomorphisms as tuples.

Definition 4.1 (Homomorphism relation)**.** Let $\mathbf{G} = (G, \lambda_G)$, $\mathbf{H} = (H, \lambda_H)$ be labeled graphs. Relation $r^{\mathbf{G}\to\mathbf{H}}$ over relation scheme $V(\mathbf{G})$ with the universe of $V(\mathbf{H})$ is called *homomorphism relation* if tuple t is present in $r^{\mathbf{G}\to\mathbf{H}}$ if and only if t is a valid homomorphism from $\mathbf{G}$ to $\mathbf{H}$.

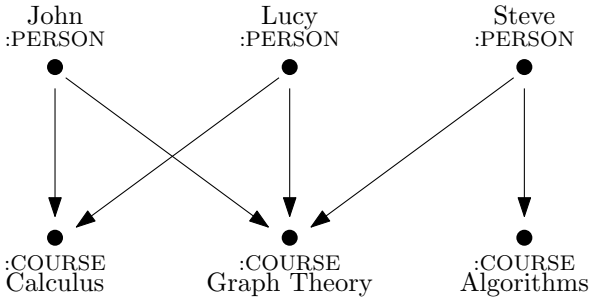
Remark. Relation $r^{\mathbf{G}\to\mathbf{H}}$ represents set of all homomorphisms $Hom(\mathbf{G}, \mathbf{H})$.

An example of a homomorphism relation for a simple graph database query is shown in Figure 4.1.

In order to evaluate homomorphisms efficiently, our interest is to use results obtained for subproblems (i.e. sets of homomorphism mappings for subgraphs of the original query graph) and assemble them in order to get the result for the original problem itself. Following theorem captures how this can be done in terms of joins.



(a) Query graph **G**



(b) Host graph **H**

v_1	v_2	v_3
John	Calculus	John
John	Calculus	Lucy
Lucy	Calculus	John
Lucy	Calculus	Lucy
John	Graph Theory	John
John	Graph Theory	Lucy
John	Graph Theory	Steve
Lucy	Graph Theory	John
Lucy	Graph Theory	Lucy
Lucy	Graph Theory	Steve
Steve	Graph Theory	John
Steve	Graph Theory	Lucy
Steve	Graph Theory	Steve
Steve	Algorithms	Steve

(c) Homomorphism relation $r^{\mathbf{G}\to\mathbf{H}}$

Figure 4.1: Homomorphism relation example

Theorem 4.1. *Let* $\mathbf{G}_1, \ldots, \mathbf{G}_k$ *be subgraphs of* $\mathbf{G}$ *such that* $\mathbf{G} = \bigcup_{i=1\ldots k} \mathbf{G}_i$*. Assume that for every graph* $\mathbf{G}_i$ *we already have the corresponding homomorphism relation* $r^{\mathbf{G}_i \to \mathbf{H}}$*. Relation* $r^{\mathbf{G} \to \mathbf{H}}$ *(and thus also the set of all homomorphisms* $Hom(\mathbf{G}, \mathbf{H})$*) can be computed as:*

$$r^{\mathbf{G} \to \mathbf{H}} = \bowtie_{i=1\ldots k} r^{\mathbf{G}_i \to \mathbf{H}}$$

Remark. Notice that the set $\{V(\mathbf{G}_i) \mid i = 1 \ldots k\}$ forms a database scheme.

Proof. We will prove that the relations $r^{\mathbf{G} \to \mathbf{H}}$ and $\bowtie_{i=1\ldots k} r^{\mathbf{G}_i \to \mathbf{H}}$ are equivalent by proving both inclusions related to these relations.

Let us start with $r^{\mathbf{G} \to \mathbf{H}} \subseteq \bowtie_{i=1\ldots k} r^{\mathbf{G}_i \to \mathbf{H}}$. Let $h \in r^{\mathbf{G} \to \mathbf{H}}$ be a homomorphism from $\mathbf{G}$ to $\mathbf{H}$. By Proposition 2.1, we can restrict our attention to an arbitrary subgraph of $\mathbf{G}$ and the appropriate restriction of h will be a homomorphism mapping from $\mathbf{G}[\cdot]$ to $\mathbf{H}$. Hence for every graph $\mathbf{G}_i$, restriction $h|_{V(\mathbf{G}_i)} \in r^{\mathbf{G}_i \to \mathbf{H}}$ must be a homomorphism from $\mathbf{G}_i$ to $\mathbf{H}$. As tuples $h|_{V(\mathbf{G}_i)}$ for every $\mathbf{G}_i$ are join–compatible, their join (equal to the mapping h) must be present in $\bowtie_{i=1\ldots k} r^{\mathbf{G}_i \to \mathbf{H}}$ which proves this inclusion.

Let us focus our attention to the second inclusion $r^{\mathbf{G} \to \mathbf{H}} \supseteq \bowtie_{i=1\ldots k} r^{\mathbf{G}_i \to \mathbf{H}}$. We will show that for any join–compatible tuples $t_1, \cdots, t_k$ from relations $r^{\mathbf{G}_1 \to \mathbf{H}}, \cdots, r^{\mathbf{G}_k \to \mathbf{H}}$, their join $t = t_1 \bowtie \cdots \bowtie t_k$ is a homomorphism from $\mathbf{G}$ to $\mathbf{H}$ (and therefore $t \in r^{\mathbf{G} \to \mathbf{H}}$). Let $e = (u, v)$ be an edge of $\mathbf{G}$. We have to show that there is an edge $e' = (t(u), t(v))$ in $\mathbf{H}$ such that $\lambda_{\mathbf{G}}(e) \subseteq \lambda_{\mathbf{H}}(e')$. We know that graphs $\mathbf{G}_i$ partition graph $\mathbf{G}$. Therefore there must be a graph $\mathbf{G}_j$ containing edge e. If edge e' was not present in $\mathbf{H}$, tuple t_j would have not been a homomorphism from $\mathbf{G}_j$ to $\mathbf{H}$ — which would have contradict our original assumption of $t_j \in r^{\mathbf{G}_j \to \mathbf{H}}$. □

The most trivial decomposition of graph $\mathbf{G}$ would be such that every $\mathbf{G}_i$ would be a graph containing a single edge of the original graph. This decomposition is however not a great choice because the resulting database scheme will not be acyclic (and therefore the evaluation of the join will be $\mathcal{NP}$-complete) unless $\mathbf{G}$ is a tree.

Let us perform this decomposition by means of computing tree decomposition. For every bag of tree decomposition, we construct graph $\mathbf{G}_i$ as an induced subgraph of $\mathbf{G}$ by the vertex set of the given bag. It is easy to see that tree decomposition can be turned into a join tree for database scheme $\{R^{\mathbf{G}_1 \to \mathbf{H}}, \cdots, R^{\mathbf{G}_k \to \mathbf{H}}\}$ induced by the tree decomposition — therefore the resulting database scheme is acyclic, allowing us to compute the join in time polynomial in the size of input and output by means of Yannakakis' algorithm. This choice however means that the computation of the homomorphism relations will be harder, actually exponential in the treewidth of $\mathbf{G}$.

Yannakakis' algorithm assumes that its input relations are available in advance which is not the case here as we have to compute the homomorphism relations first. In the remainder of this chapter, we will present an algorithm tailored to this specific problem

of computing $Hom(\mathbf{G}, \mathbf{H})$. Knowing the goal behind the computation of homomorphism relations, additional properties (like the consistency of the database) may be enforced on the go and additional information that makes the joining procedure more straightforward can be kept.

The general outline of our algorithm will be similar to the computation of $Hom(\mathbf{G}, \mathbf{H})$ by means of Yannakakis' algorithm. Let us list main steps needed for successful application of Yannakakis' algorithm on this problem:

1. Compute homomorphism relation for every node of tree decomposition.
2. Compute full reduction of the database.
3. Evaluate the project–join query.

4.1 Database construction algorithm

The major part of our algorithm deals with computation of the database of homomorphism relations and ensuring that the resulting database is globally consistent. This is equivalent to first two steps in the outline described above.

In order to simplify the derivation of the algorithm, we will consider a special case of tree decomposition. This decomposition will allow us to focus on atomic operations of addition and removal of a single vertex.

Definition 4.2 (Nice tree decomposition)**.** Rooted tree decomposition is called *nice tree decomposition* if for every (non-root) node u and its parent node v one of the following hold:

- $\chi(u) \subseteq \chi(v)$ and $|\chi(u)| = |\chi(v)| - 1$. In such case u is called *forget node*.
- $\chi(v) \subseteq \chi(u)$ and $|\chi(u)| = |\chi(v)| + 1$. In such case u is called *introduce node*.

Remark. To grasp better intuition about forget and introduce nodes we will provide an informal description of these types of nodes. Node u is called forget node, if one vertex got forgotten on the last edge of the path from root to u. Similarly node u is an introduce node, if one vertex was introduced on the last edge of the path from root to u.

Every tree decomposition can be turned to a nice one by a simple algorithm. Let us start by rooting the original tree decomposition in an arbitrary vertex.

First possibility how the property of nice tree decomposition can be violated is that $\chi(n_i) = \chi(n_j)$ for some node n_i and its parent n_j. This problem can be easily fixed by removing n_i from the tree decomposition and attaching all its children to n_j.

Another problem arises when multiple vertices get forgotten and introduced when traversing an edge. To recover from this issue, we have to replace this single edge by

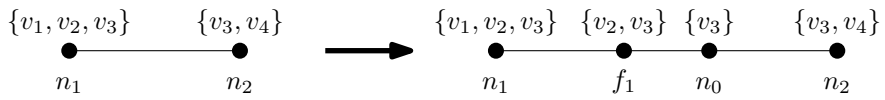


Figure 4.2: Computing nice tree decomposition

a sequence of edges so that these forget and introduce operations are applied one by one. To keep the size of the bags as small as possible, vertices are first forgotten. An application of this principle is shown in Figure 4.2. As sizes of bags are bounded by $tw + 1$ where tw is the treewidth, an edge in the tree decomposition is replaced by at most $2tw + 2$ edges.

Similarly as in the case of backtracking algorithm, our algorithm will be working with partial solutions — although in this case these will not be directly partial homomorphisms. In this case a partial solution will be represented by a partial database — database containing portion of homomorphism relations.

Definition 4.3 (Partial solution)**.** Let $\mathbf{G} = (G, \lambda_G)$ be the query graph, $\mathbf{H} = (H, \lambda_H)$ be the host graph. Let (T, χ) be a nice tree decomposition of G rooted in r. Pair (S, d) is a *partial solution* of $Hom(\mathbf{G}, \mathbf{H})$ with respect to (T, χ) if:

- S is a connected subset of nodes of T.
- S contains root r.
- Database scheme of d is the set of bags of nodes in S, i.e.

  $$D = \{\chi(n) \mid n \in S\}$$

  Remark. This means that for every node $n \in S$, there is a relation r_n over relation scheme $\chi(n)$ present in the database.
- Let $V_S = \bigcup_{n \in S} \chi(n)$ be a set of all vertices of $\mathbf{G}$ covered by nodes in S. The set of all homomorphisms from $\mathbf{G}[V_S]$ to $\mathbf{H}$ is equal to the join of the database d, i.e.:

  $$Hom(\mathbf{G}[V_S], \mathbf{H}) = \bowtie_{r \in d} r$$

- For every node $n \in S$, relation $r_n \in d$ over relation scheme $\chi(n)$ is the smallest one such that the above property holds.

We can see that partial solutions are indeed closely related to the solution of this part of the algorithm. Partial solution represents a globally consistent database of homomorphism relations for a subproblem described by a connected subtree of the tree decomposition.

Corollary 4.1. *Let (S, d) be a partial solution. If $S = V(T)$, the partial solution (S, d) represents the solution to the original problem of $Hom(\mathbf{G}, \mathbf{H})$.*

Proof. Tree decomposition has to cover all nodes of the graph, i.e. $\bigcup_{n \in V(T)} \chi(n) = V(\mathbf{G})$. The induced subgraph $\mathbf{G}[V_S]$ is therefore equal to $\mathbf{G}$, as is $Hom(\mathbf{G}[V_S], \mathbf{H})$ equal to $Hom(\mathbf{G}, \mathbf{H})$. □

Figure 4.3 shows a pseudocode for the database construction algorithm. For the sake of clarity, some important points were omitted from the pseudocode and will be addressed in the following text.

The initial partial solution has to cover root node. There are several ways to compute the relation associated to the root node. One possibility is to apply a conventional algorithm for computing homomorphisms (e.g. bactracking algorithm as discussed in Figure 2.14). Another option involves choosing a specific root node, e.g. a node with an empty bag when the associated relation contains just one empty tuple. It is always possible to modify a tree decomposition to obtain such a node.

The choice of node n in every iteration of the main loop of the algorithm is a nondeterministic choice. From the theoretical point of view, the actual choice is not important — but in practice choosing n properly may lead to results in shorter time. We will discuss heuristics for choosing n in Chapter 5.

The last hidden point of the algorithm is the way it ensures global consistency of the database in case that for some $t \in r_p$ no join–compatible tuple exists in r_n. This procedure is roughly captured in Figure 4.3 starting on line 16, but it needs more explanation.

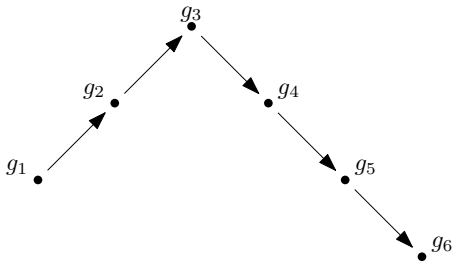
Alongside with the generation of tuples (both on lines 9 and 14), it is possible to construct a *consistency graph*. Vertices of this graph correspond to individual tuples in the relations of the database. Edge between tuples $t_m \in r_m$ and $t_n \in r_n$ is present in the graph if t_m and t_n are join–compatible and importantly nodes m and n are adjacent in T. It is easy to check that an edge connects t_m with t_n if and only if t_m was generated from t_n (or vice versa). The structure of this graph is shown in Figure 4.4.

Global consistency of the database means, that for every tuple $t \in r_{n_i}$ and for every node $n_j \in S$ adjacent to n_i, there is an edge connecting t with some tuple from r_{n_j}. As this property is local, a removal of a tuple may render this property violated only in the nearest neighborhood of t. The consistency can be reestablished by traversing this complementary graph structure as shown in Figure 4.5.

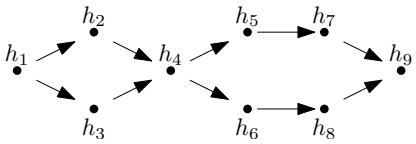
Require: query graph $\mathbf{G}$, host graph $\mathbf{H}$, nice tree decomposition (T, χ) of $\mathbf{G}$
Ensure: globally consistent database of homomorphism relations for the problem of computing $Hom(\mathbf{G}, \mathbf{H})$

```
1:
2:  function ConstructDatabase( )
3:      (S, d) ← initial partial solution
4:      while S ≠ V(T) do
5:          n ← node from V(T) such that n ∉ S and nodes in S ∪ {n} are connected
6:          p ← parent node of n (note that p ∈ S)
7:
8:          if n is a forget node then
9:              r_n ← π_{χ(n)}(r_p)
10:         else if n is an introduce node (introducing vertex v) then
11:             r_n ← {}
12:             for all t ∈ r_p do
13:                 if ∃v' ∈ V(H) such that (t ∪ {v → v'}) ∈ Hom(G[χ(n)], H) then
14:                     For every such v', extend t by an assignment v → v' and add it to r_n
15:                 else
16:                     r_p ← r_p \ {t}
17:
18:                     remove all tuples from relations of d that lost their only join
19:                     compatible tuple in r_p after deletion of t; cascade
20:                 end if
21:             end for
22:         end if
23:
24:         (S, d) ← (S ∪ {n}, d ∪ {r_n})
25:     end while
26:     return database d
27: end function
```

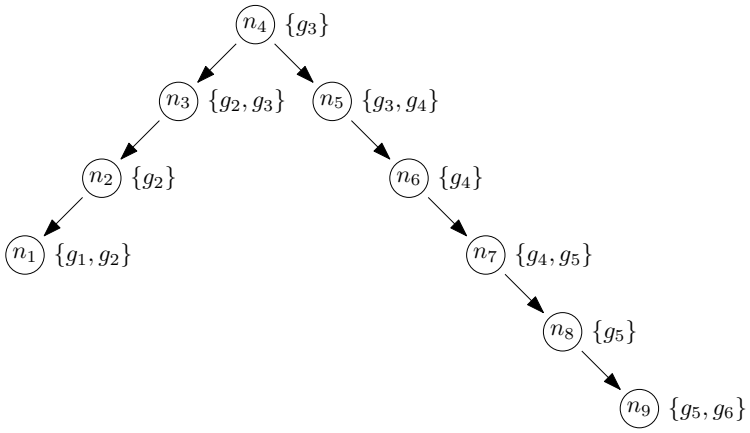
Figure 4.3: Database construction algorithm



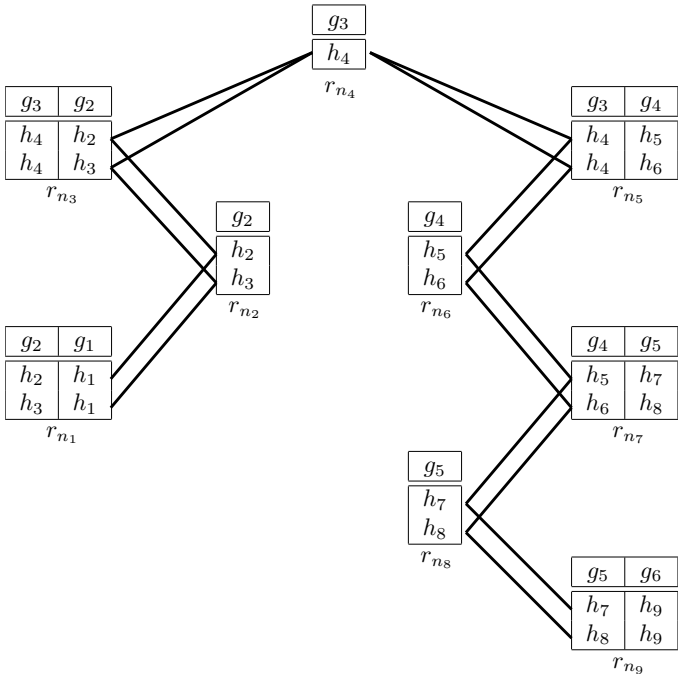
(a) Query graph **G**



(b) Host graph **H**



(c) Nice tree decomposition of **G**



(d) Example of a solution with a consistency graph

Figure 4.4: Example of a solution with a consistency graph

```
1: procedure REMOVE(n, t)
2:     remove tuple t from relation r_n
3:     for all tuples t' ∈ r_n' adjacent to t in the consistency graph do
4:         remove edge {t, t'} from the consistency graph
5:         if {t, t'} was the only edge connecting t' with tuples in r_n then
6:             REMOVE(n', t')
7:         end if
8:     end for
9: end procedure
```

Figure 4.5: Consistency ensuring algorithm

Proposition 4.1. *After application of the* REMOVE *procedure on a globally consistent database, the database is in a consistent state.*

Proof. It follows from global consistency and the definition of the consistency graph that for every two adjacent nodes $x, y \in S$, every tuple $t \in r_x$ must be connected to some tuple in r_y (as otherwise this tuple would have been inconsitent). Firstly we will show that the existence of such edges is also sufficient for global consistency. Let us assume that such edges are present in the consistency graph, i.e. every tuple is connected to some tuple in every adjacent relation.

It is obvious that for every two adjacent nodes $x, y \in S$, relations r_x and r_y are consistent. For the consistency of two relations, it is sufficient that every tuple $t \in r_x$ finds it join–compatible tuple $t' \in r_y$ (and vice versa) — i.e. there is an edge $\{t, t'\}$ in the consistency graph, which was our assumption. Adjacent nodes are connected by a path of unit length.

Let us assume that for every two nodes u, v of T connected by a path of length at most $i-1$, relations r_u and r_v are consistent. Let us show that in such case, nodes x, y of T connected by a path of length at most i are consistent as well. If $i > 1$, we can find a node z on path from x to y such that both x, z and y, z are connected by paths of length at most $i-1$. Hence relations r_x, r_z and r_y, r_z are consistent. As $\chi(x) \cap \chi(y) \subseteq \chi(z)$, relations r_x and r_y are consistent as well.

Every two nodes of tree decomposition are connected by a path, hence the database is pairwise consistent. As tree decomposition induces an acyclic database scheme, the database is also in a globally consistent state.

An edge is removed from the consistency graph only when the REMOVE procedure removes one of its endpoints. In such a case this procedure inspects the other endpoint as well and checks whether required edges are still present in the consistency graph. It is therefore impossible that an inconsistent tuple was left in the database and the resulting database therefore must be consistent.

□

Theorem 4.2. *After every iteration of the main loop of the algorithm from Figure 4.3, (S, d) contains a valid partial solution.*

Proof. The algorithm starts with a valid partial solution. We have to check that each iteration of the main loop transforms one partial solution into another one. Let (S, d) be a partial solution before entering body of this loop. We will check that $(S \cup \{n\}, d \cup \{r_n\})$ is once again a partial solution. Let us check that it satisfies all the required properties.

It is obvious that set $S \cup \{n\}$ contains root node r. Root r was present in the initial solution and since then only new nodes were added. Similarly it is easy to check that nodes in $S \cup \{n\}$ are connected in T as this was required on line 5. Adding relation r_n over relation scheme $\chi(n)$ to the database ensures that the database is over a proper database scheme.

It remains to verify two last properties. Let us start by verifying that the relations of $d \cup \{r_n\}$ are minimal — i.e. there are no tuples that are not join–compatible with adjacent relations (with respect to T). The consistency between r_n and r_p is obvious: If n is a forget node then for every tuple $t \in r_p$ there is a join–compatible tuple $\pi_{\chi(n)}(t)$ present in r_n, and similarly every tuple of r_n must have been projected from some tuple of r_p. If n is an introduce node, then for every tuple $t \in r_p$ either a tuple t in r_n was generated (that agrees with t on all the attributes from $\chi(p)$), or tuple t was removed from r_p. The consistency of remaining relations of d is ensured by the Remove procedure. Correctness of this procedure was proved separately.

Let $V_{S'} = V_S \cup \chi(n)$. Let us check that the newly constructed database represents all homomorpshims from $\mathbf{G}[V_{S'}]$ to $\mathbf{H}$. Let us consider the forget node case first. In such a case, $V_S = V_{S'}$, no tuple was removed from the original database d (and its join is equal to $Hom(\mathbf{G}[V_S], \mathbf{H})$). Relations r_n and r_p are consistent and relation scheme of r_p is a superset of the relation scheme of r_n. Therefore $r_n \bowtie r_p = r_p$.

$$\begin{aligned}(\bowtie_{r \in d} r) \bowtie r_n &= (\bowtie_{r \in d} r) \bowtie (r_p \bowtie r_n) \\ &= (\bowtie_{r \in d} r) \bowtie r_p \\ &= \bowtie_{r \in d} r = Hom(\mathbf{G}[V_S], \mathbf{H}) = Hom(\mathbf{G}[V_{S'}], \mathbf{H})\end{aligned}$$

Let us discuss the introduce node case now. All homomorpshims from $\mathbf{G}[\chi(n)]$ to $\mathbf{H}$ consistent with r_p are present in r_n — this was ensured on line 14. The removal procedure then removed only inconsistent tuples that does not participate on the join anyway. The resulting database is therefore full reduction of a database of homomorphism relations for the decomposition $\{\mathbf{G}[\chi(m)] \mid m \in S \cup \{n\}\}$. □

Proposition 4.2. *Algorithm from the listing in Figure 4.3 terminates.*

Proof. Tree decomposition is a connected graph. Therefore if $S \neq V(T)$, there is a node $n \notin S$ that is connected to some node from S. In every iteration, one node is added to the set S and after $|V(T)| - 1$ iterations the algorithm terminates. □

Corollary 4.2. *Algorithm from the listing in Figure 4.3 is correct.*

Proof. Algorithm reaches the situation when $S = V(T)$ while ensuring that (S, d) is a partial solution. According to the Corollary 4.1, the join of the database d is equal to $Hom(\mathbf{G}, \mathbf{H})$ as was required. □

4.1.1 Complexity

This section devoted to the time complexity analysis of the database construction algorithm is divided into three parts. In the first one, necessary data structures and algorithms will be presented. In the second part, the problem of computing valid extensions for a given partial homomorphism is discussed. The third part is devoted to the analysis of the database construction algorithm itself.

Preliminaries

Many structures used in our algorithm are sets — and set operations have to be performed efficiently. We will start by discussing the way how sets are represented in our algorithm and what does it mean for the time needed to compute set operations like intersection and union.

Let U be a finite universe. We say that set S is a *set over universe* U if $S \subseteq U$. If two sets S_1, S_2 are over the same universe U, their intersection $S_1 \cap S_2$ and union $S_1 \cup S_2$ are once again sets over the universe U.

We will assume that universe U is a totally ordered set. Set S over the universe U will be represented as a sorted list containing elements of U in increasing order. No element of U can be present in the list more than once.

Operations of intersection and union of two sets will be performed by a 2–way merge algorithm. We assume that the reader is familiar with this algorithm, nevertheless a brief overview of this algorithm will be given.

We will describe the application of 2–way merge for computing an intersection of two sets S_1, S_2 over the same universe U. The idea is simple:

$$S_1 \cap S_2 = \begin{cases} \varnothing & S_1 = \varnothing \vee S_2 = \varnothing \\ (S_1 \setminus \{\min(S_1)\}) \cap S_2 & \min(S_1) < \min(S_2) \\ S_1 \cap (S_2 \setminus \{\min(S_2)\}) & \min(S_1) > \min(S_2) \\ \{\min(S_1)\} \cup \Big((S_1 \setminus \{\min(S_1)\}) \cap (S_2 \setminus \{\min(S_2)\})\Big) & \min(S_1) = \min(S_2) \end{cases}$$

The operations of finding the minimum of a set and removal of the smallest element are done in constant time for the sorted list representation — the minimum in a sorted list is the first element of the list and its removal is done by truncating the list (i.e. advancing to the next element). For every two sets S_1, S_2 over the universe U the algorithm computes the intersection in time $\mathcal{O}(c \cdot |U|)$ where $\mathcal{O}(c)$ is the time needed to compare two elements of U. The idea is similar for computing a union of two sets.

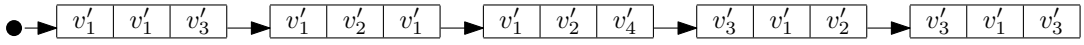
The 2–way merge procedure can be used to merge multiple sets over the same universe U. As every merge of two sets over U is once again a set over U, the merge of n sets can be done in time $\mathcal{O}(n \cdot c \cdot |U|)$. This procedure of merging n sets over the same universe will be called n–way merge.

Let us now discuss the representation of a graph $\mathbf{H}$. When talking about vertices, set $V(\mathbf{H})$ will be treated as a sorted universe — for simplicity we will assume that two vertices $v_i, v_j \in V(\mathbf{H})$ are compared $v_i \leq v_j$ if and only if $i \leq j$. Any subset of $V(\mathbf{H})$ is therefore a set over the universe of $V(\mathbf{H})$ and is represented by a sorted list.

For every vertex v of $\mathbf{H}$ an *incidence list* will be known. This list contains all edges e of $\mathbf{H}$ such that either $e = (v, u)$ or $e = (u, v)$ (i.e. all edges incident to v). The incidence list for vertex v will be denoted by $inc(v)$.

When we will be working with some subset of $E(\mathbf{H})$, it will always be a subset of some incidence list. For this reason in the context of edges, $inc(v)$ will be seen as an universe. An order of edges in $inc(v)$ has to be therefore established. Let $end_v(e)$ denote endpoint of edge e other than v (we will often call such endpoint *opposing*). Two edges $e_i, e_j \in inc(v)$ are compared first by the opposing vertex (with respect to the order on $V(\mathbf{H})$) secondly by the edge itself — more formally:

$$compare_v(e_i, e_j) = \begin{cases} e_i < e_j & \text{if } end_v(e_i) < end_v(e_j) \\ e_i < e_j & \text{if } end_v(e_i) = end_v(e_j) \wedge i < j \\ e_i > e_j & \text{if } end_v(e_i) > end_v(e_j) \\ e_i > e_j & \text{if } end_v(e_i) = end_v(e_j) \wedge i > j \\ e_i = e_j & \text{otherwise} \end{cases}$$



(a) List representation

v'_1	v'_1	v'_3
v'_1	v'_2	v'_1
v'_1	v'_2	v'_4
v'_3	v'_1	v'_2
v'_3	v'_1	v'_3

(b) Tabular representation

Figure 4.6: Representation of a relation over $[v_1, v_2, v_3]$

This allows not only efficient merging of two sets over the universe $inc(v)$. If set $S \subseteq inc(v)$ of edges is represented by a sorted list, it is possible to obtain a list of opposing endpoints (i.e. set $\{end_v(e) \mid e \in S\}$) that is ordered with respect to the ordering on $V(\mathbf{H})$ by a single pass of S. As $inc(v)$ is ordered firstly by the opposing vertex, duplicate vertices are aligned in blocks — which allows the deduplication to be done efficiently.

Another important set–related structures used in the database construction algorithm are relations. In order to derive the overall complexity of this algorithm, the complexity of individual operations related to them has to be studied. Only relations with the domains of $V(\mathbf{H})$ are used in our algoirhtm — therefore set $V(\mathbf{H})$ will be used in the text below instead of a general universe U.

Let us fix the order $A_1, \cdots, A_k$ of attributes in a k–attribute relation scheme R. Tuples over R need not then be represented as mappings $t : R \to V(\mathbf{H})$ but rather as k–tuples $(t(A_1), \cdots, t(A_k))$. This allows us to establish lexicographical order on such k–tuples and set $\mathcal{R}_k$ of all such k–tuples then forms an ordered universe in the sense of our view of sets. It is easy to see that universe $\mathcal{R}_k$ contains $|V(\mathbf{H})|^k$ elements. A k–attribute relation r is then a set over the universe $\mathcal{R}_k$. A list based representation of such a relation is shown in Figure 4.6a, but it is also convenient to view such a list as a table shown in Figure 4.6b. To emphasize the fact that the relation scheme is ordered, we will use notation $[A_1, \cdots, A_k]$ instead of $\{A_1, \cdots, A_k\}$.

Two operations related to relations have to be considered. When processing an introduce node a $(k-1)$–attribute relation r_p over relation scheme $[v_1, \cdots, v_{k-1}]$ is considered and k–attribute relation r_n over relation scheme $[v_1, \cdots, v_{k-1}, v_k]$ has to be computed (i.e. a column has to be appended at the end of r_p). Assume that for every tuple $t \in r_n$ a sorted list $ext(t)$ of values for attribute v_k is known (and for every v'_k from this list, tuple $t \cup \{v_k \to v'_k\}$ has to be present in r_n). Let us iterate through all tuples $t = (v'_1, \cdots, v'_{k-1})$ of r_p and for each value v'_k from the list $ext(t)$ construct a tuple $t' = (v'_1, \cdots, v'_{k-1}, v'_k)$. As both r_p and all $ext(t)$ are sorted, it is easy to see that tuples t' are generated in lexicographical order (and thus rendering the relation r_n properly sorted). At most $|V(\mathbf{H})|^{k-1}$ tuples from relation r_p

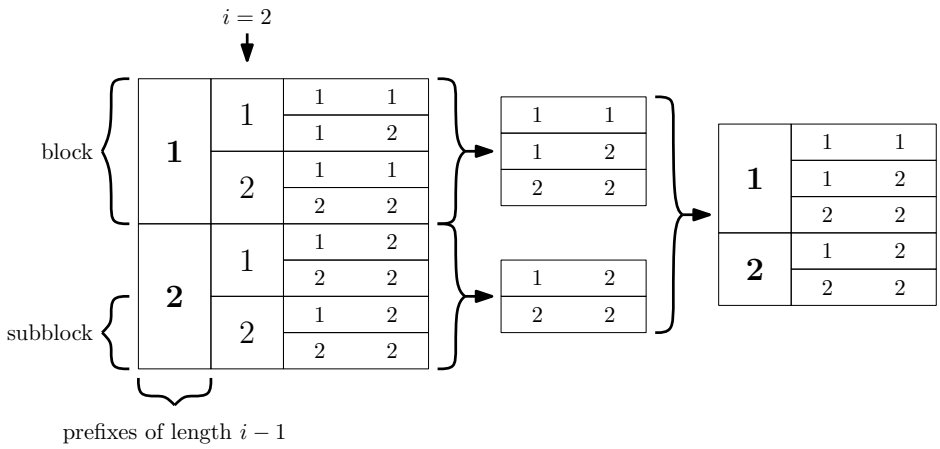


Figure 4.7: Removal of a column

have to be considered and at most $|V(\mathbf{H})|^k$ tuples are generated (each having $\mathcal{O}(k)$ elements) — thus the time of appending a column to a $(k-1)$–attribute relation is in $\mathcal{O}(k \cdot |V(\mathbf{H})|^k)$.

In the case of forget nodes, relation r_n is obtained by projecting a k–attribute relation r_p onto $k-1$ attributes — i.e. a column is removed from a table. Let us assume that i–th column is about to be removed. If tuple $(x_1, \cdots, x_{i-1}, x_i, x_{i+1}, \cdots, x_k)$ is present in r_p, tuple $(x_1, \cdots, x_{i-1}, x_{i+1}, \cdots, x_k)$ has to be present in r_n. The following procedure may seem unintuitive at the first glance — reader may therefore consult Figure 4.7 where the individual steps are illustrated. Let us identify blocks of tuples having the same prefix of length $i-1$ — these blocks can be handled separately as the results for individual blocks can be joined to form a relation in lexicographical order. There are up to $|V(\mathbf{H})|^{i-1}$ such blocks. Each of the blocks can be decomposed into subblocks having the same value in the i–th column — there are up to $|V(\mathbf{H})|$ subblocks for each block. Suffixes of tuples in subblocks of length $k-i$ (from the universe $\mathcal{R}_{k-i}$) are clearly in lexicographical order — this allows us to compute their union by means of $|V(\mathbf{H})|$–way merge in time $\mathcal{O}(|V(\mathbf{H})| \cdot k \cdot |V(\mathbf{H})|^{k-i})$ (assuming that two suffixes are compared in $\mathcal{O}(k)$ time). The overall time complexity of the removal of a column from a k–attribute relation is therefore $\mathcal{O}(k \cdot |V(\mathbf{H})|^k)$.

Extending partial homomorphism tuples

Let us recall how an introduce node (introducing vertex v) is processed by the database construction algorithm. Every tuple t from parent's relation r_p is considered and vertices v' such that $t' = t \cup \{v \to v'\}$ forms a valid homomorphism from $\mathbf{G}[\chi(n)]$ to $\mathbf{H}$ are found. For all such vertices v', tuple t' is added to the newly constructed relation r_n. The way relations are represented allows to perform this operation efficiently provided set $ext(t)$ of all such vertices v' is known for every tuple $t \in r_p$. Let us discuss how vertices v' can be found for a single partial homomorphism t and hence construct set $ext(t)$.

Let $t = \{u_1 \to u'_1, \cdots, u_k \to u'_k\}$ be a partial homomorphism from $\mathbf{G}$ to $\mathbf{H}$. We are supposed to find all vertices v' of $\mathbf{H}$ such that $t \cup \{v \to v'\}$ is once again a partial homomorphism. Not all vertices v' are allowed — the choice of these vertices is restricted by two types of constraints originating from graph $\mathbf{G}$.

Firstly, vertex v of $\mathbf{G}$ has labels — and these labels have to be present for vertex v' of $\mathbf{H}$ as well (i.e. $\lambda_{\mathbf{G}}(v) \subseteq \lambda_{\mathbf{H}}(v')$ must hold). The set of vertices v' satisfying this condition will be denoted by $label(v)$.

Secondly, the assignment for vertex v has to respect edges connecting v with vertices $u_1, \cdots, u_k$ (for which the assignment is already known). Let us focus on a single edge $e = (u_i, v)$ of $\mathbf{G}$ labeled by $\lambda_{\mathbf{G}}(e)$. Edge e *admits* assignment $v \to v'$ given assignment $u_i \to u'_i$ if there is an edge $e' = (u'_i, v')$ such that $\lambda_{\mathbf{G}}(e) \subseteq \lambda_{\mathbf{H}}(e')$. Set of all vertices v' of $\mathbf{H}$ for which the assignment $v \to v'$ is admitted by edge e will be denoted by $adm(e, u_i \to u'_i)$. The meaning of this set is similar if edge e is oppositely directed — in such case edge e' has to be oppositely directed as well.

Vertex v' can be used to extend partial homomorphism t by an assignment $v \to v'$ if it satisfies all constraints of both types, i.e. if vertex v' is present in all sets related to these constraints. The set of all such vertices can be computed by means of intersection:

$$label(v) \cap \left[\bigcap_{i=1\ldots k} \left(\bigcap_{e \in E_{\mathbf{G}}(u_i, v)} adm(e, u_i \to u'_i) \right) \right]$$

(where $E_{\mathbf{G}}(x, y)$ denotes the set of all edges of $\mathbf{G}$ connecting vertices x and y)

The problem of this formula is, that the number of intersected sets depends on the number of edges in $\mathbf{G}$ connecting vertex v with some of the vertices $u_1, \cdots, u_k$ (i.e. the size of sets $E_{\mathbf{G}}(u_i, v)$). We can avoid it by precomputing the inner intersection for every vertex of $\mathbf{G}$:

$$edge_v(u_i \to u'_i) = \bigcap_{e \in E_{\mathbf{G}}(u_i, v)} adm(e, u_i \to u'_i)$$

Let us provide intuitive meaning of this set. If vertex v' is present in the set $edge_v(u_i \to u'_i)$, no edge in $\mathbf{G}$ prevents the assignment $v \to v'$ when the assignment $u_i \to u'_i$ is already fixed. Introducing these sets allows a simplification of the formula to one using the intersection of $k + 1$ sets:

$$label(v) \cap \left[\bigcap_{i=1\ldots k} edge_v(u_i \to u'_i) \right]$$

The precomputation of sets $label(\cdot)$, $adm(\cdot)$ and $edge_{\cdot}(\cdot)$ does not come for free. The time required to compute these sets is discussed below.

Proposition 4.3. *The time needed to compute sets $label(v)$ for all vertices v of $\mathbf{G}$ is in $\mathcal{O}(L_V \cdot |V(\mathbf{H})| + |V(\mathbf{G})|)$ (where $L_V = \sum_{v \in V(\mathbf{G})} |\lambda_{\mathbf{G}}(v)|$). Resulting sets $label(v)$ are represented by sorted lists ordered with respect to the ordering on $V(\mathbf{H})$.*

Proof. Let us consider the time needed to compute $label(v)$ for a single vertex v first. If $\lambda_{\mathbf{G}}(v)$ is empty, $label(v)$ contains all vertices of $\mathbf{H}$ — thus $label(v)$ can be computed in constant time.

If $\lambda_{\mathbf{G}}(v)$ is non–empty, every vertex v' of $\mathbf{H}$ has to be considered and the property $\lambda_{\mathbf{G}}(v) \subseteq \lambda_{\mathbf{H}}(v')$ has to be verified (if it is satisfied, vertex v' should be present in $label(v)$). Assuming that presence of a single label in $\lambda_{\mathbf{H}}(v')$ can be done in constant time, the inclusion property is verified by $|\lambda_{\mathbf{G}}(v)|$ such queries. The overall time to compute $label(v)$ for a given vertex v is therefore in $\mathcal{O}(1 + |\lambda_{\mathbf{G}}(v)| \cdot |V(\mathbf{H})|)$. Let us now sum over all vertices of $\mathbf{G}$ to obtain the total time needed to construct sets $label(v)$ for all vertices v of $\mathbf{G}$:

$$\begin{aligned}
\sum_{v \in V(\mathbf{G})} \mathcal{O}\Big(1 + |\lambda_{\mathbf{G}}(v)| \cdot |V(\mathbf{H})|\Big) &= \mathcal{O}\left(\sum_{v \in V(\mathbf{G})} \Big[1 + |\lambda_{\mathbf{G}}(v)| \cdot |V(\mathbf{H})|\Big]\right)^{1} \\
&= \mathcal{O}\left(|V(\mathbf{G})| + |V(\mathbf{H})| \sum_{v \in V(\mathbf{G})} |\lambda_{\mathbf{G}}(v)|\right) \\
&= \mathcal{O}\Big(|V(\mathbf{G})| + |V(\mathbf{H})| \cdot L_V\Big)
\end{aligned}$$

As $V(\mathbf{H})$ is represented by an ordered list of vertices, its subsets $label(v)$ are represented by sorted lists as well. □

Proposition 4.4. *The time needed to compute sets $adm(e, u \to u')$ for every edge e of $\mathbf{G}$, every its endpoint u and every vertex u' of $\mathbf{H}$ is in $\mathcal{O}((L_E + |E(\mathbf{G})|) \cdot |E(\mathbf{H})|)$ (where $L_E = \sum_{e \in E(\mathbf{G})} |\lambda_{\mathbf{G}}(e)|$). Lists representing sets $adm(e, u \to u')$ are sorted with respect to the ordering on $V(\mathbf{H})$.*

Proof. Let us consider the time needed to compute set $adm(e, u \to u')$ for a fixed edge e of $\mathbf{G}$, its endpoint u and a fixed vertex u' of $\mathbf{H}$. Vertex v' is in $adm(e, u \to u')$ if there is an edge $e' = (u', v')$ (respectively (v', u') if e has opposite direction) such that $\lambda_{\mathbf{G}}(e) \subseteq \lambda_{\mathbf{H}}(e')$. All such edges are incident to u', hence present in the incidence list $inc(u')$. Every edge in the incidence list $inc(u')$ is considered. If $\lambda_{\mathbf{G}}(e)$ is empty, only direction of the edge has to

[1] If $f_1(n) \in \mathcal{O}(g_1(n))$ and $f_2(n) \in \mathcal{O}(g_2(n))$ then $f_1(n) + f_2(n) \in \mathcal{O}(g_1(n) + g_2(n))$. To make the proofs more readable we abuse the notation and write directly $\mathcal{O}(g_1(n)) + \mathcal{O}(g_2(n)) = \mathcal{O}(g_1(n) + g_2(n))$.

be verified — hence $\mathcal{O}(1)$ time per edge. If $\lambda_{\mathbf{G}}(e)$ is non–empty, every label $l \in \lambda_{\mathbf{G}}(e)$ has to be checked to be present in $\lambda_{\mathbf{H}}(e')$ using $|\lambda_{\mathbf{G}}(e)|$ constant time queries.

We are not interested in edges $e' = (u', v')$ — set $adm(e, u \to u')$ has to contain opposing vertices v'. Recall that the incidence list $inc(u')$ is sorted by vertex v', thus it is possible to convert set of edges e' into set of vertices v' (and remove duplicate endpoints v') by a single pass through the list of found edges. The overall time needed to compute a single set $adm(e, u \to u')$ is therefore in $\mathcal{O}(|inc(u')| \cdot |\lambda_{\mathbf{G}}(e)| + |inc(u')|)$.

Let us now aggregate this result over all edges e of $\mathbf{G}$, both its endpoints and all vertices u' of $\mathbf{H}$:

$$
\begin{aligned}
&2 \sum_{e \in E(\mathbf{G})} \sum_{u' \in V(\mathbf{H})} \mathcal{O}\Big(|inc(u')| \cdot (|\lambda_{\mathbf{G}}(e)| + 1)\Big) \\
&\qquad = \sum_{e \in E(\mathbf{G})} \mathcal{O}\left((|\lambda_{\mathbf{G}}(e)| + 1) \sum_{u' \in V(\mathbf{H})} |inc(u')| \right) \\
&\qquad = \sum_{e \in E(\mathbf{G})} \mathcal{O}\Big((|\lambda_{\mathbf{G}}(e)| + 1) \cdot |E(\mathbf{H})| \Big) \quad {}^{2} \\
&\qquad = \mathcal{O}\Big(|E(\mathbf{H})| \cdot \sum_{e \in E(\mathbf{G})} (|\lambda_{\mathbf{G}}(e)| + 1) \Big) \\
&\qquad = \mathcal{O}\Big(|E(\mathbf{H})| \cdot L_E + |E(\mathbf{H})| \cdot |E(\mathbf{G})| \Big)
\end{aligned}
$$

It is easy to see that the lists representing sets $adm(e, u \to u')$ are sorted with respect to the ordering on $V(\mathbf{H})$ as the incidence lists used to compute these sets were sorted by opposing vertices of edges. □

Proposition 4.5. *Provided sets $adm(\cdot)$ are known, sets $edge_y(x \to x')$ for every vertices x, y of $\mathbf{G}$ and every vertex x' of $\mathbf{H}$ are computed in $\mathcal{O}(|E(\mathbf{G})| \cdot |E(\mathbf{H})| + |V(\mathbf{G})|^2 \cdot |V(\mathbf{H})|)$.*

Proof. Let us study time needed to compute a single set $edge_y(x \to x')$. If there is no edge connecting vertices x and y in $\mathbf{G}$, then every vertex y' of $\mathbf{H}$ is in $edge_y(x \to x')$ and $edge_y(x \to x')$ is computed in constant time.

If there are some edges connecting vertices x and y in $\mathbf{G}$, sets $adm(e, x \to x')$ for every edge e connecting x and y are to be intersected. Recall that sets $adm(e, x \to x')$ were found out by inspecting adjacency list $inc(x')$ — hence $adm(e, x \to x')$ are sets represented by sorted lists over a universe of size at most $|inc(x')|$. The intersection is computed by

[2] Using handshaking lemma (Euler): $\sum_{v \in V} \deg(v) = 2|E|$ (where $\deg(v)$ denotes degree of vertex v, i.e. size of the incidence list $inc(v)$)

means of N_{xy}–way merge (where N_{xy} is number of edges connecting x and y in $\mathbf{G}$) in time $\mathcal{O}(N_{xy} \cdot |inc(x')|)$. The overall complexity of computing $edge_y(x \to x')$ for fixed vertices x, y of $\mathbf{G}$ and x' of $\mathbf{H}$ is in time $\mathcal{O}(N_{xy} \cdot |inc(x')| + 1)$.

Let us now sum over vertices x, y of $\mathbf{G}$ and vertices x' of $\mathbf{H}$ to obtain time needed to compute all sets $edge_{\cdot}(\cdot)$:

$$\begin{aligned}
&\sum_{x,y \in V(\mathbf{G})} \sum_{x' \in V(\mathbf{H})} \mathcal{O}(N_{xy} \cdot |inc(x')| + 1) \\
&\quad = \sum_{x,y \in V(\mathbf{G})} \mathcal{O}\left(|V(\mathbf{H})| + N_{xy} \cdot \sum_{x' \in V(\mathbf{H})} |inc(x')| \right) \\
&\quad = \sum_{x,y \in V(\mathbf{G})} \mathcal{O}\left(|V(\mathbf{H})| + N_{xy} \cdot |E(\mathbf{H})| \right) \\
&\quad = \mathcal{O}\left(|V(\mathbf{G})|^2 \cdot |V(\mathbf{H})| + |E(\mathbf{H})| \cdot \sum_{x,y \in V(\mathbf{G})} N_{xy} \right) \\
&\quad = \mathcal{O}\left(|V(\mathbf{G})|^2 \cdot |V(\mathbf{H})| + |E(\mathbf{H})| \cdot |E(\mathbf{G})| \right)
\end{aligned}$$

□

Database construction algorithm complexity

Theorem 4.3. *Given sets $label(\cdot)$ and $edge_{\cdot}(\cdot)$, the database construction algorithm runs in time $\mathcal{O}(|V(T)| \cdot \mathbf{tw}(\mathbf{G}) \cdot |V(\mathbf{H})|^{\mathbf{tw}(\mathbf{G})+1})$.*

Proof. In the course of the database construction algorithm, three types of nodes are being processed — the root node, forget nodes and introduce nodes. Let us consider the latter two first.

If n is a forget node and p is the parent node, a $(|\chi(p)| - 1)$–attribute relation r_n is created from a $|\chi(p)|$–attribute relation r_p. The way relations are represented allows the computation of such a relation to be done in time $\mathcal{O}(|\chi(p)| \cdot |V(\mathbf{H})|^{|\chi(p)|})$.

When processing an introduce node n introducing vertex v, a $|\chi(n)|$–attribute relation r_n is constructed from a $(|\chi(n)|-1)$–attribute relation r_p. This is done by means of the algorithm for appending a column to the representation of relation r_p in time $\mathcal{O}(|\chi(n)| \cdot |V(\mathbf{H})|^{|\chi(n)|})$ — provided set $ext(t)$ is known for every tuple t of r_p. We have shown in previous part that a single set $ext(t)$ can be computed using $|\chi(n)|$–way merge of sets over the universe $V(\mathbf{H})$ as:

$$ext(t) = label(v) \cap \left[\bigcap_{u \in \chi(p)} edge_v\Big(u \to t(u)\Big) \right]$$

There are at most $|V(\mathbf{H})|^{|\chi(n)|-1}$ tuples in r_p and the computation of the set $ext(t)$ for a single tuple takes $\mathcal{O}(|\chi(n)| \cdot |V(\mathbf{H})|)$ time — hence all sets $ext(t)$ can be computed in time $\mathcal{O}(|\chi(n)| \cdot |V(\mathbf{H})|^{|\chi(n)|})$ which is also the overall time needed to process the introduce node n.

Finally let us consider the time needed to compute the relation associated to the root node r. Let us construct a sequence of nodes $n_0, \cdots, n_{|\chi(r)|}$ such that $\chi(n_0) = \varnothing$, $\chi(n_{|\chi(r)|}) = \chi(r)$ and $\chi(n_i) \subset \chi(n_{i+1})$. It is easy to see that $\chi(n_i)$ contains exactly i vertices and n_i can be seen as an introduce node with respect to n_{i-1}. Relation r_{n_0} is trivially constructed in constant time, while $\mathcal{O}(i \cdot |V(\mathbf{H})|^i)$ time is needed to compute a relation associated to the introduce node n_i. All relations associated to nodes $n_0, \cdots, n_{|\chi(r)|}$ are computed in time $\sum_{i=1}^{|\chi(r)|} \mathcal{O}(i \cdot |V(\mathbf{H})|^i)$, which is in $\mathcal{O}(|\chi(r)| \cdot |V(\mathbf{H})|^{|\chi(r)|})$. As root relation r_r is equal to $r_{n_{|\chi(r)|}}$, the root relation can be computed in time $\mathcal{O}(|\chi(r)| \cdot |V(\mathbf{H})|^{|\chi(r)|})$ as well.

For every node n of the tree decomposition, size of the set $\chi(n)$ is bounded by $\mathbf{tw}(\mathbf{G}) + 1$. This allows us to process all types of nodes in time $\mathcal{O}(\mathbf{tw}(\mathbf{G}) \cdot |V(\mathbf{H})|^{\mathbf{tw}(\mathbf{G})+1})$. There are $|V(T)|$ nodes to be processed, hence the algorithm runs in $\mathcal{O}(|V(T)| \cdot \mathbf{tw}(\mathbf{G}) \cdot |V(\mathbf{H})|^{\mathbf{tw}(\mathbf{G})+1})$.

It remains to verify that ensuring the database consistency does not worsen this bound. An edge in the consistency graph is created in situations when a tuple in some of the relations is generated. We know that every edge in this graph is traversed at most once during the whole run of the database construction algorithm (as it is removed after it is being traversed). Hence both creation and maintenance of the consistency graph is linear in the number of tuples generated by the algorithm. □

4.2 Result generating algorithms

In this section, we will focus on three algorithms that provided output of previous algorithm answer one of the following questions:

- Does homomorphism from $\mathbf{G}$ to $\mathbf{H}$ exist? (decision variant)
- How many homomorphisms from $\mathbf{G}$ to $\mathbf{H}$ exist? (counting variant)
- What are the homomorphisms from $\mathbf{G}$ to $\mathbf{H}$? (enumeration variant)

We will see that compared to the algorithm presented in previous section, these algorithms are very simple. This is mainly caused by the presence of the consistency graph.

4.2.1 Decision algorithm

The procedure of deciding whether a homomorphism from $\mathbf{G}$ to $\mathbf{H}$ exist is so simple that calling it an algorithm may well be unnecesary. We have to decide if the join of the database obtained from the database construction algorithm generates at least one tuple.

It suffices to check whether the database contains at least one tuple in any of the relations — as the database is globally consistent, every tuple must participate on the join and thus presence of a tuple in the database means that the join is non–empty. The algorithm runs therefore in constant time.

4.2.2 Counting algorithm

In this subsection a dynamic programming algorithm for counting number of distinct homomorphisms from **G** to **H** will be proposed. This algorithm will perform a bottom–up traversal of the tree decomposition and the tuples associated to these nodes.

Let $\mathrm{X}(n)$ denote the set of all vertices covered by the subtree of n, i.e. $\mathrm{X}(n)$ is the union of bags of all nodes in n's subtree. By $c_{n,t}$ we will denote the number of homomorphisms from $\mathbf{G}[\mathrm{X}(n)]$ to **H** that are compatible with tuple $t \in r_n$. We will derive a formula that allows computation of $c_{n,t}$ in bottom–up manner.

Let $n_1, \cdots, n_k$ be children of n and JC_{t,n_i} be the set of join–compatible tuples of t in relation r_{n_i} (these can be obtained from the consistency graph). Notice that for any $j \neq k$, any two tuples $t_j \in JC_{t,n_j}$, $t_k \in JC_{t,n_k}$ are join–compatible (this is caused by the fact that any vertex in $\chi(n_j) \cap \chi(n_k)$ must be present also in $\chi(n)$ — and the assignments for these vertices are fixed by tuple t). This fact allows us to use multiplication to compute $c_{n,t}$:

$$c_{n,t} = \prod_{i=1\ldots k} \sum_{t' \in JC_{t,n_i}} c_{n_i,t'}$$

After having computed all $c_{n,t}$, it remains to sum over all tuples of the relation assigned to the root node to obtain the result:

$$\sum_{t \in r_r} c_{r,t}$$

Every edge of the consistency graph is used exactly once during the computation of $c_{n,t}$, hence the counting algorithm runs in linear time in the size of the graph of join–compatible tuples (i.e. linear in the number of tuples in the database).

4.2.3 Enumeration algorithm

The enumeration algorithm uses similar principle as the counting one, except that instead of numbers of homomorphisms $c_{n,t}$ it computes sets of all homomorphisms from $\mathbf{G}[\mathrm{X}(n)]$ to **H** compatible with tuple t, denoted by $H_{n,t}$. The formula for computation of $H_{n,t}$ is slightly more complex in this case:

$$H_{n,t} = \{t'_1 \bowtie \cdots \bowtie t'_k \mid \forall t_1 \in JC_{t,n_1}, \cdots, t_k \in JC_{t,n_k} : t'_i \in H_{n_i,t_i}\}$$

One has to be aware of a special case when all children of node n are forget nodes (forgetting the same vertex v). In such a case we have to enrich homomorphisms by the assignment for vertex v from tuple t, i.e.:

$$H_{n,t} = \{\{v \to t(v)\} \bowtie t'_1 \bowtie \cdots \bowtie t'_k \mid \forall t_1 \in JC_{t,n_1}, \cdots, t_k \in JC_{t,n_k} : t'_i \in H_{n_i,t_i}\}$$

To obtain the result, it suffices to perform the union of $H_{r,t}$ for every tuple of the relation associated to the root node r:

$$\bigcup_{t \in r_r} H_{r,t}$$

Proposition 4.6. *Number of tuples generated by enumeration algorithm for instance* $(\mathbf{G}, \mathbf{H})$ *is in* $\mathcal{O}(|V(T)| \cdot |Hom(\mathbf{G}, \mathbf{H})|)$.

Proof. Let us focus on a single node n of the tree decomposition and its associated sets $H_{n,t}$. Let $H_n = \bigcup_{t \in r_n} H_{n,t}$ denote all homomorphisms from $\mathbf{G}[X(n)]$ to $\mathbf{H}$. Sets $H_{n,t}$ for $t \in r_n$ are pairwise disjoint, hence $|H_n| = \sum_{t \in r_n} |H_{n,t}|$.

As the database is consistent, every mapping $h \in H_n$ is used to construct some homomorphism mapping from $\mathbf{G}$ to $\mathbf{H}$ — the number of tuples $|H_n|$ generated when processing node n of T is therefore bounded by $|Hom(\mathbf{G}, \mathbf{H})|$. There are $|V(T)|$ nodes to be processed, hence at most $|V(T)| \cdot |Hom(\mathbf{G}, \mathbf{H})|$ tuples are generated in the course of the algorithm. □

CHAPTER 5

Experimental results

In previous chapter, theoretical guarantees for our algorithm were given. This chapter will be devoted to its experimental evaluation.

To simulate real world datasets, test instances used in this chapter will be generated from Barabàsi–Albert model[1] (unless stated otherwise). Many real world networks are thought to respect scale–free power–law distribution of vertex degrees — and the Barabàsi–Albert model generates such graphs. In power–law graphs, the number of vertices of degree k decays exponentially in k.

At every step a new vertex is created. This new vertex is then connected to at most K vertices that were formerly present in the graph. Vertices with high degree are more likely to be chosen (preferential attachment property). The parameter K will be referred to as the max–links parameter.

5.1 Comparison with Neo4j

For the following experiments, the latest version of Neo4j available at the moment of writing the thesis (Neo4j 2.2.1 Community Edition) was used. Unfortunately during our experiments, it turned out that this version does not always return correct results. In many situations it either misses some valid homomorphisms or it even constructs a solution that is not a valid homomorphism. This behavior was detected by the `Neo4jTest` experiment from the attached software bundle. Figure 5.1 shows an instance where Neo4j produced an invalid homomorphism:

$$\{q_0 \to h_2, q_1 \to h_1, q_2 \to h_3, q_4 \to h_0\}$$

(a) Query graph **G** **(b)** Host graph **H**

Figure 5.1: Instance where Neo4j failed

Following Cypher query was used to obtain this result:

```
match (q0)-->(q1), (q0)-->(q2), (q1)-->(q2), (q3)-->(q0), (q3)-->(q2)
return q0.name, q1.name, q2.name, q3.name
```

The problems can be emphasized by comparing results of previous query with an equivalent query:

```
match (q0), (q1), (q2), (q3)
where (q0)-->(q1) and (q0)-->(q2) and (q1)-->(q2) and (q3)-->(q0)
                  and (q3)-->(q2)
return q0.name, q1.name, q2.name, q3.name
```

This query is highly inefficient (query execution plan suggests that it is solved by testing all 5^4 possible mappings) — but in this case it correctly returns just one result:

$$\{q_0 \to h_0, q_1 \to h_2, q_2 \to h_1, q_4 \to h_4\}$$

Due to the inefficiency of the second query, queries of the first type were used in the experiments. Reader should be aware that once the bug is fixed, the running times of Neo4j may change.

5.1.1 Scalability in the size of G

Graph database queries are typically rather simple (in terms of the treewidth). This subsection studies the scalability of the algorithms in the size of **G** when the treewidth of **G** is kept small. Let us start this subsection by artificially constructed instances introduced in Figure 2.15. Regardless of the length of the cycle in graph **G** and the length of the double chain in **H**, no homomorphism from **G** to **H** exists. This allows us to study the performance of the algorithm in its pure form without spending time on enumerating results.

In this experiment, graphs **G** are directed cycles of increasing length while graph **H** is similar to the one shown in Figure 2.15, except that the graph was prolonged — the longest

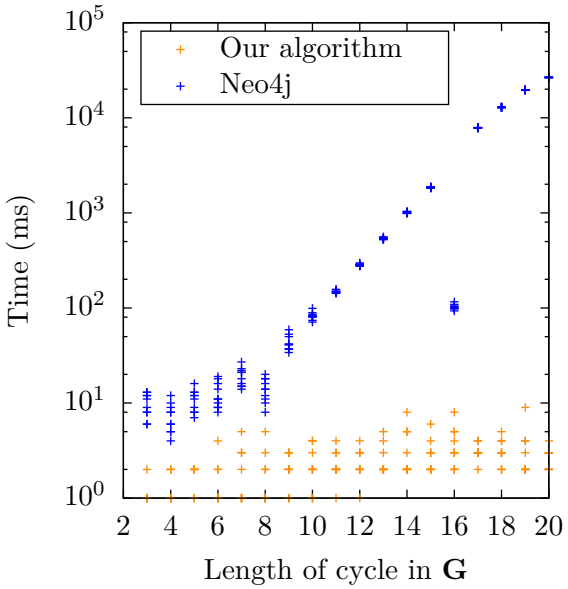


Figure 5.2: Scalability in the size of **G** (artificially constructed instances)

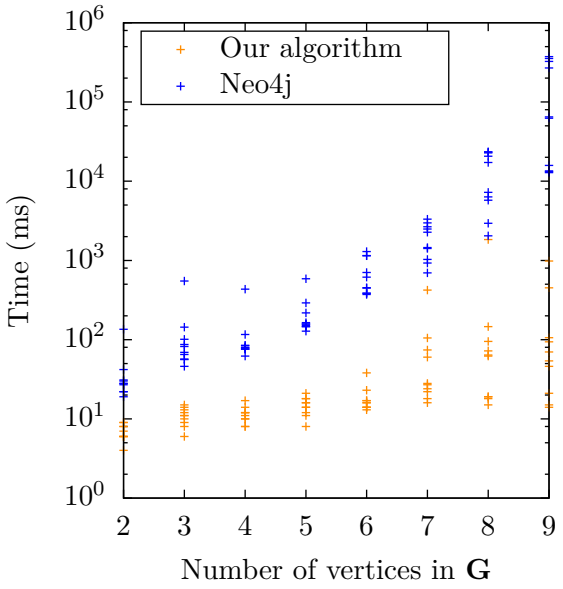


Figure 5.3: Scalability in the size of **G**

directed path in graph **H** is 20 edges long. In Figure 5.2 time needed to enumerate results for different lengths of cycle in **G** is shown. The graph suggests that the runtime of Neo4j is growing exponentially in the size of **G** while runtime of our algorithm is polynomial — which agrees with the results from the previous chapter.

To see how these algorithms scale in more real situations, another experiment was conducted. Graphs **G** were generated from Barabàsi-Albert model with max–links set to 2 (only graphs **G** with no non–trivial automorphisms were considered). Graphs **H** were generated with max–links set to 5 ($|V(\mathbf{H})| = 300$). Measured runtime is plotted in Figure 5.3. The trend of fast–growing runtimes of Neo4j (compared with our algorithm) is less pronounced, but still apparent.

It has to be said that the situation when the queries are fixed and the database (i.e. graph **H**) is continually growing is a more typical scenario in practice. The scalability of the algorithms in the size of **H** will be discussed in the next subsection.

5.1.2 Scalability in the size of **H**

In the following set of experiments a random graph **G** was generated for every run (and this choice of **G** was then fixed). Graphs **H** of increasing size were then used and the time needed to enumerate $Hom(\mathbf{G}, \mathbf{H})$ was measured.

Let us start the discussion with graphs **G** of treewidth 1 — i.e. trees. Acyclic queries are the most typical queries in practical scenarios. Figures 5.4 and 5.5 shows measured runtimes

and number of homomorphisms returned for tree query graphs **G** with 4 and 8 vertices. Graphs **H** were generated from Barabàsi–Albert model with max–links set to 4.

In both cases, our algorithm tends to return results faster than Neo4j — however Neo4j runtimes seem to be longer just by a constant factor. This could be attributed to different operating conditions of the algorithms — Neo4j as a full–fledged database engine uses a HDD to store the graph (in our case we have employed a SSD drive), whereas our algorithm is operating in memory. Another common theme in both cases is that Neo4j fails to recover some valid homomorphism mappings — all homomorphism mappings returned by our algorithm were verified to be correct.

Whereas in the case of 4–vertex query graph, the database construction algorithm consumed significant part of the total runtime, in the case of larger query graph its contribution to the total runtime became negligible. This has two reasons: Firstly the number of solutions significantly increased in case of the larger query graph, secondly the addition of vertices to **G** while keeping the same treewidth does worsen the database construction algorithm runtime only by a constant factor.

Let us perform a similar experiment with more complex query graphs. Figure 5.6 captures results measured for instances where Barabàsi–Albert model with max–links set to 3 was used to generate query graph **G** and graphs **H** were generated from Barabàsi–Albert model with max–links set to 6. Query graph **G** contains 8 vertices.

In some cases Neo4j returns results even before the database construction part of our algorithm terminates — this holds especially for large graphs **H**. On the other hand, Figure 5.6b shows that Neo4j misses most of the results discovered by our algorithm and furthermore majority of the mappings constructed by Neo4j are not valid homomorphisms.

So far we have discussed results that were highly influenced by the time needed to enumerate homomorphisms. This time has to be considered — but most of the real world databases and queries are designed in such a way that the number of results tends to be low.

Let us now focus on the database construction algorithm separately. This algorithm forms the most critical part of the overall algorithm — after it terminates a constant time is required to enumerate each of the results. In the following two experiments graphs **H** were generated from Barabàsi–Albert model with max–links set to 6.

Figure 5.7a shows measured runtimes for query graphs with treewidth 1 (i.e. trees). We have approximated the runtimes by a power function. From the theoretical discussion in Chapter 4 we know that the exponent of the power functions should be below 2 and this holds for all three approximating functions.

Figure 5.7b captures results of a similar experiment with query graphs **G** generated from Barabàsi–Albert model with max–links set to 2. The result may seem strange — the exponents of the approximating functions are lower than in the previous case even though query graphs **G** are more complex this time. The explanation of this phenomenon is

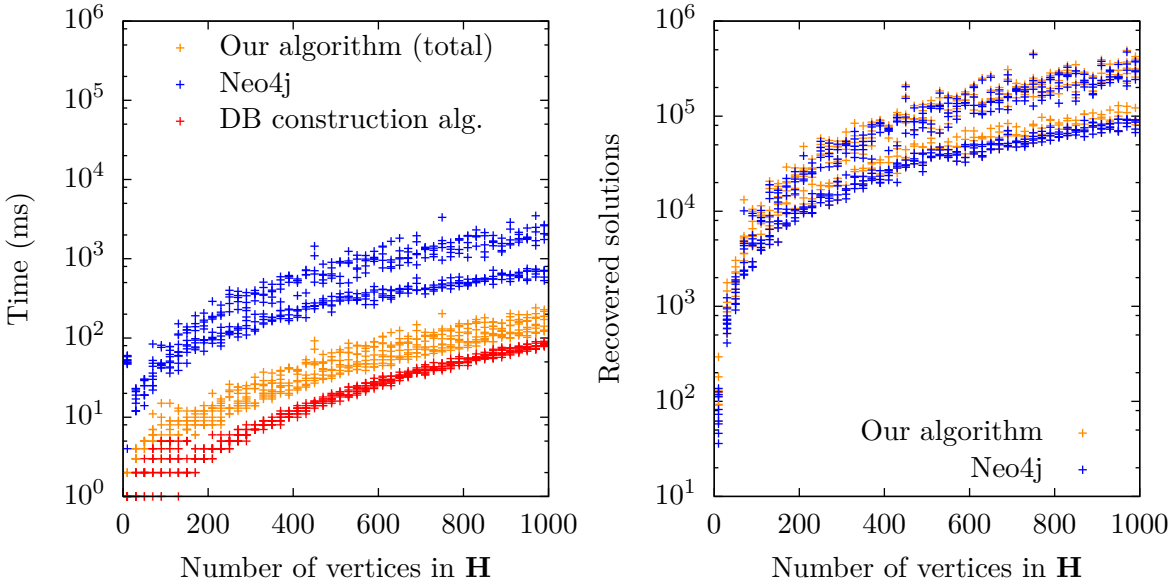


Figure 5.4: Scalability in the size of $\mathbf{H}$ — $\mathbf{tw}(\mathbf{G}) = 1$, $|V(\mathbf{G})| = 4$

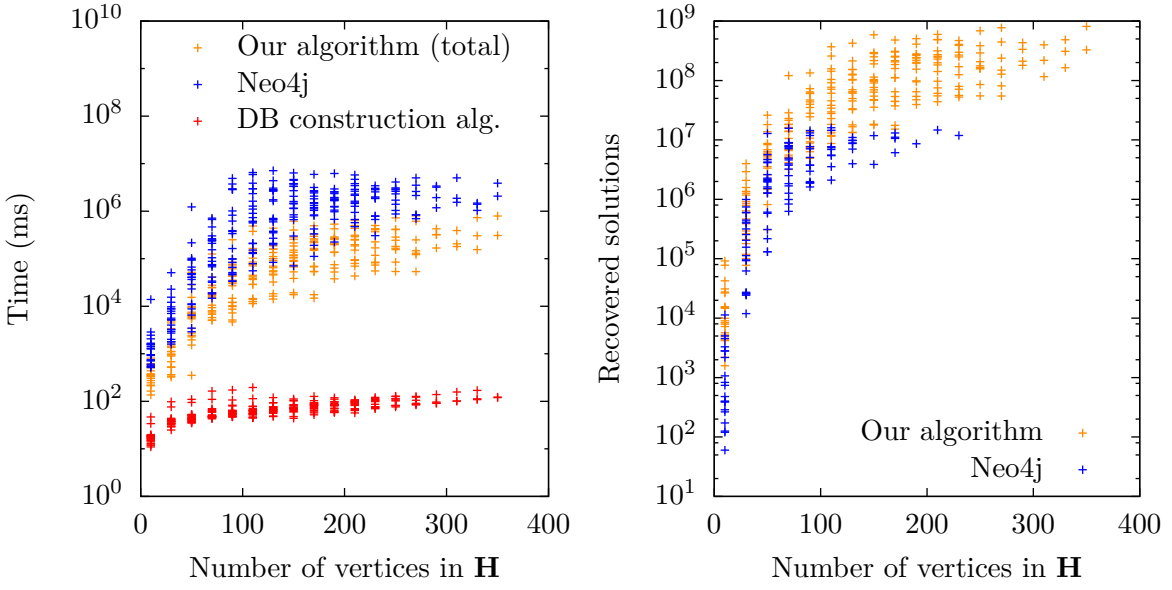
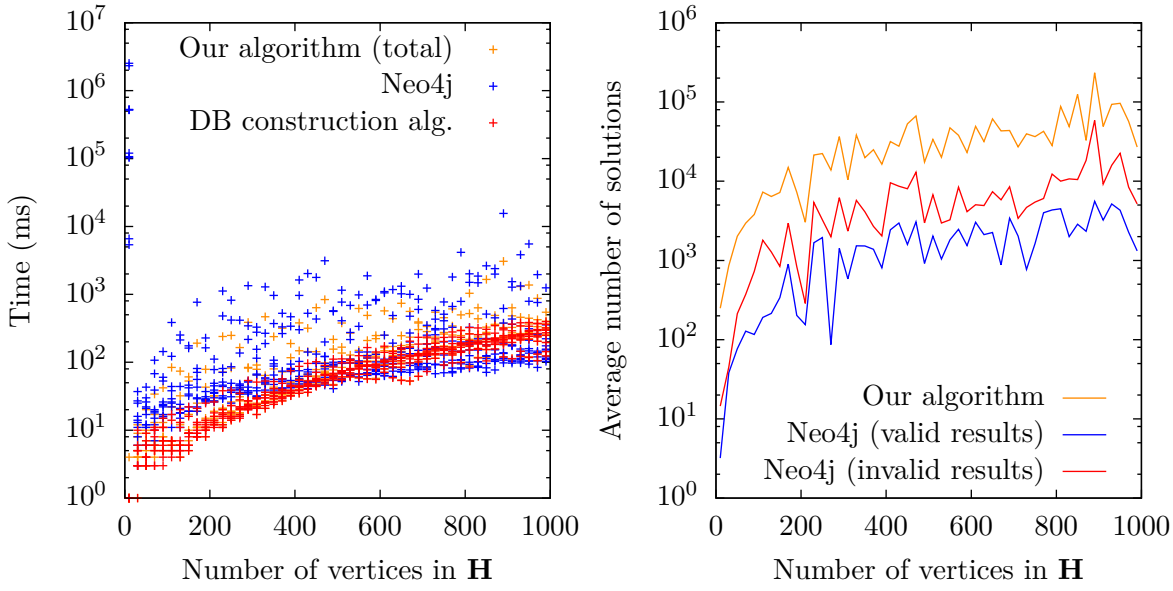


Figure 5.5: Scalability in the size of $\mathbf{H}$ — $\mathbf{tw}(\mathbf{G}) = 1$, $|V(\mathbf{G})| = 8$



(a) Runtime **(b)** Generated homomorphisms

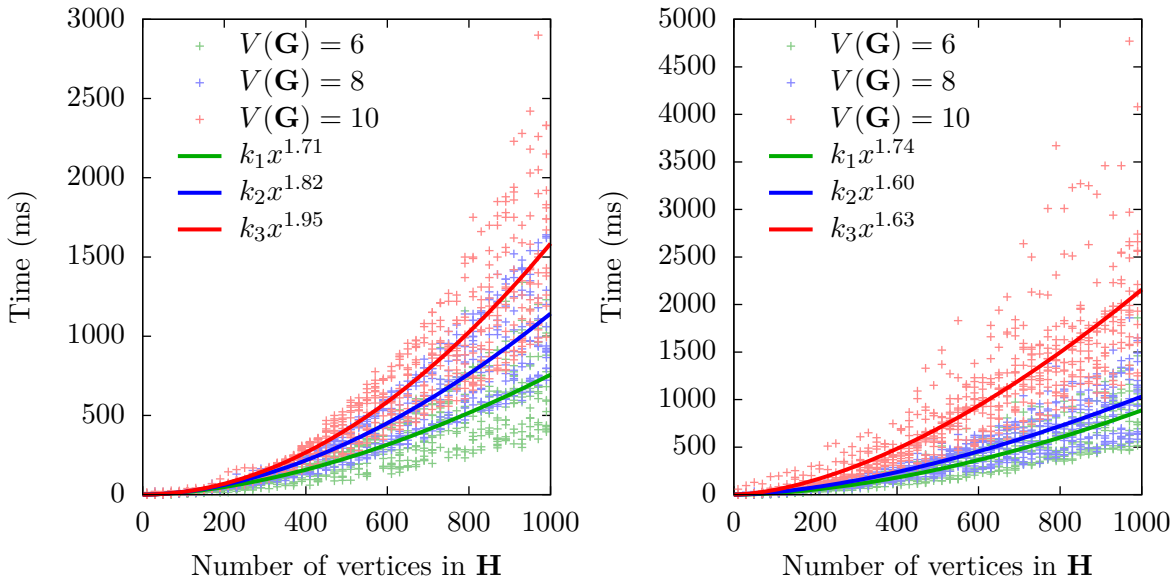
Figure 5.6: Scalability in the size of **H** — $|V(\mathbf{G})| = 8$, Barabàsi–Albert, max–links=3

however straightforward. More complex graphs **G** are more restrictive. This results in faster identification of inconsistent tuples and smaller relations throughout the computation.

5.2 Effect of planning

The goal of database engines is to produce correct results in the shortest time possible. Typically one cannot rely on a single algorithm and single settings for every instance. Every instance has its specific parameters which allows sophisticated systems to choose proper algorithms to solve the task, or at least fine–tune their internals.

Usually there is not a single way to evaluate a query. The first thing successful database engines do (after understanding the semantics of a query) is that they try to collect relevant information to decide which algorithms to apply, in which order and what settings to use — they construct a *query execution plan*. This plan prescribes what exactly has to be done in order to yield the result. Neo4j uses a planner for evaluating queries as well — an example of a query execution plan of Neo4j is shown in Figure 5.8. These planners commonly consider even low–level operations to estimate the runtime — e.g. time required to perform disk operations[19] — and statistical information about the state of the database is used. The actual choice of the best plan is often a hard problem and hence heuristics come to play.



(a) G — Barabàsi–Albert, max–links=1 **(b) G** — Barabàsi–Albert, max–links=2

Figure 5.7: Database construction algorithm — scalability in the size of **H**

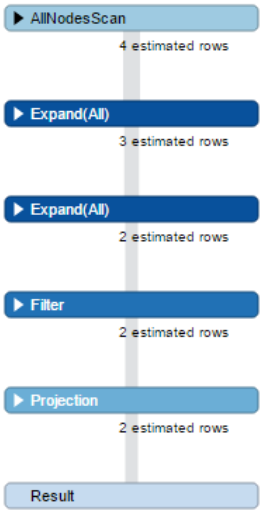


Figure 5.8: Neo4j query plan

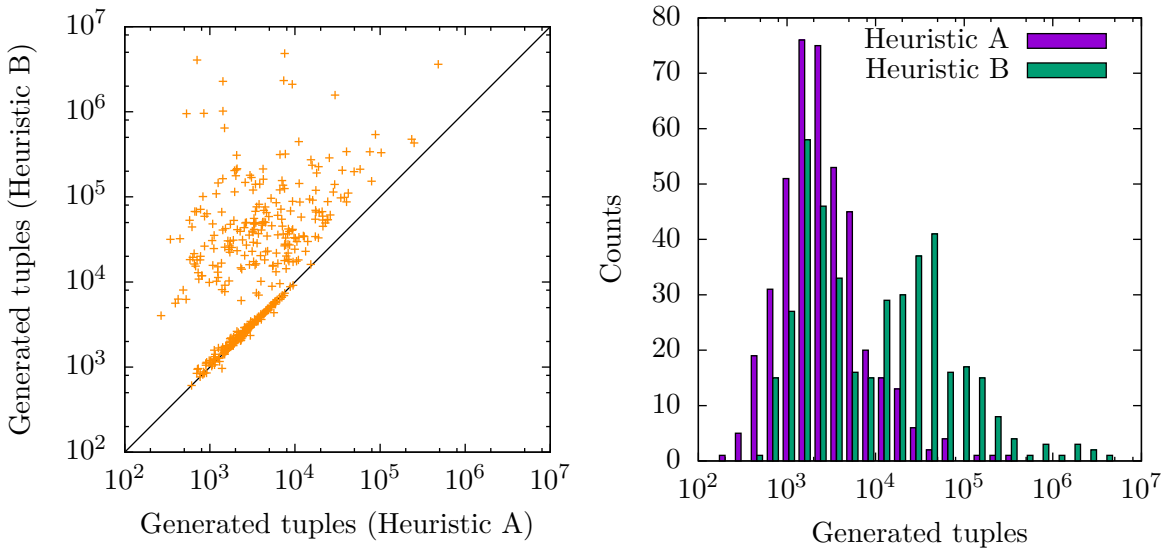
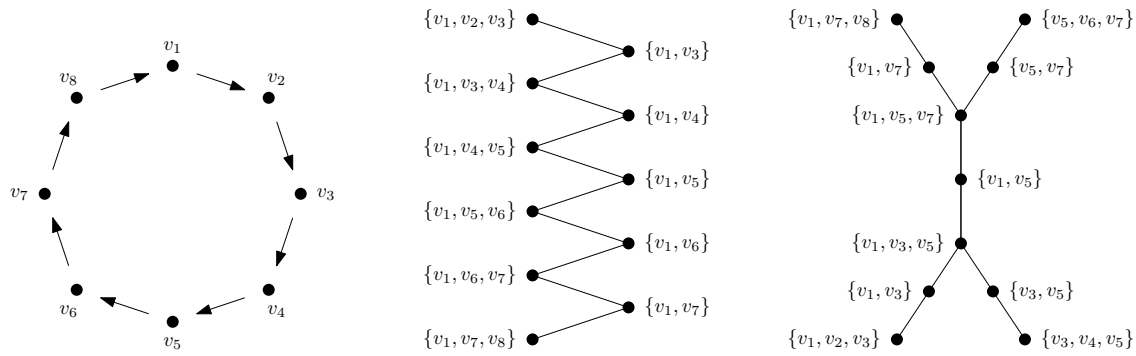


Figure 5.9: Effect of heuristical node expansion

Small details may often have significant impact on the query execution time — and this holds for the algorithm from Chapter 4 as well. In the database construction algorithm (Figure 4.3), expansion nodes n in the main loop of the algorithm are chosen in a non-deterministic way. Choosing these nodes in proper order may result in much lower amount of generated tuples throughout the course of the algorithm — and therefore also in significantly shorter runtimes. Generating new tuples from a tuple that will be later found out to be inconsistent is a waste of time. Hence it is a good idea to try to detect this inconsistency in as early stage as possible.

Let us demonstrate this effect on a simple heuristic. It is convenient to introduce constraints as soon as possible — and the only constraints in our case are edges and labels in $\mathbf{G}$. At every expansion step, node n introducing the most edges is chosen (i.e. a node that maximizes the number of edges between the vertex newly introduced and the vertices that are in the parent's bag). This also means that introduce nodes are processed before forget nodes (if they are available). The heuristics will also be applied to choose the root for the nice tree decomposition — a node will be chosen as a root if it admits the lexicographically largest sequence of the numbers of edges introduced.

To empirically measure the impact of this heuristic, following experiment was conducted. Random graphs $\mathbf{G}$ and $\mathbf{H}$ were generated from the Barabàsi–Albert model. For every instance, the database construction algorithm was run twice using different heuristics and the number of generated tuples was counted. The first heuristic (referred to as Heuristic A) was described in the previous paragraph. The second heuristic (Heuristic B) aims to achieve exactly the opposite goal — the sequence it produces is the lexicographically smallest



(a) Graph **G** **(b)** Linear decomposition of **G** **(c)** Bad decomposition of **G**

Figure 5.10: Possible tree decompositions of a directed cycle

one. According to Figure 5.9, the proposed heuristic is not perfect (in some instances this heuristic performed even worse than the Heuristic B), but in most of the cases the number of generated tuples was significantly lower compared to Heuristic B.

We have shown that in many cases a proper ordering of expansion nodes can result in significant improvement of the runtime. There are however cases when even choosing the optimal order results in suboptimal performance because the underlying tree decomposition was chosen inappropriately.

This issue will be demonstrated on the artificial instances from Subsection 5.1.1. Figure 5.10 shows a query graph **G** (a directed cycle of length 8) together with its two optimal tree decompositions (the width of both of them is 2). After rooting, these decompositions satisfy all properties of a nice tree decomposition. Let us consider number of generated tuples for the double chain graph **H** with $2L$ vertices for both of these tree decompositions — results are shown in Figure 5.11.

This result may seem strange at the first glance — tree decompositions from Figure 5.10 share many common properties. They have the same number of nodes and the number of nodes with equally sized bags is the same in both tree decompositions as well. But the important fact is that these decompositions differ structurally. Clearly it does not make sense to root the tree decomposition from Figure 5.10c in any of the internal nodes — the bags of the internal nodes are disconnected in **G** and the number of tuples in the initial relation would be at least quadratic in L. Without loss of generality, let us root it in the node with bag $\{v_1, v_2, v_3\}$. The problem with this decomposition is that once the introduce node with bag $\{v_1, v_3, v_5\}$ is processed, the newly introduced vertex v_5 lacks any edges to v_1 and v_3 — there are no constraints on the newly introduced vertex. As the number of tuples in the parent relation is linear in L and for every tuple $2L$ extensions are possible, the newly constructed relation will be quadratic in size.

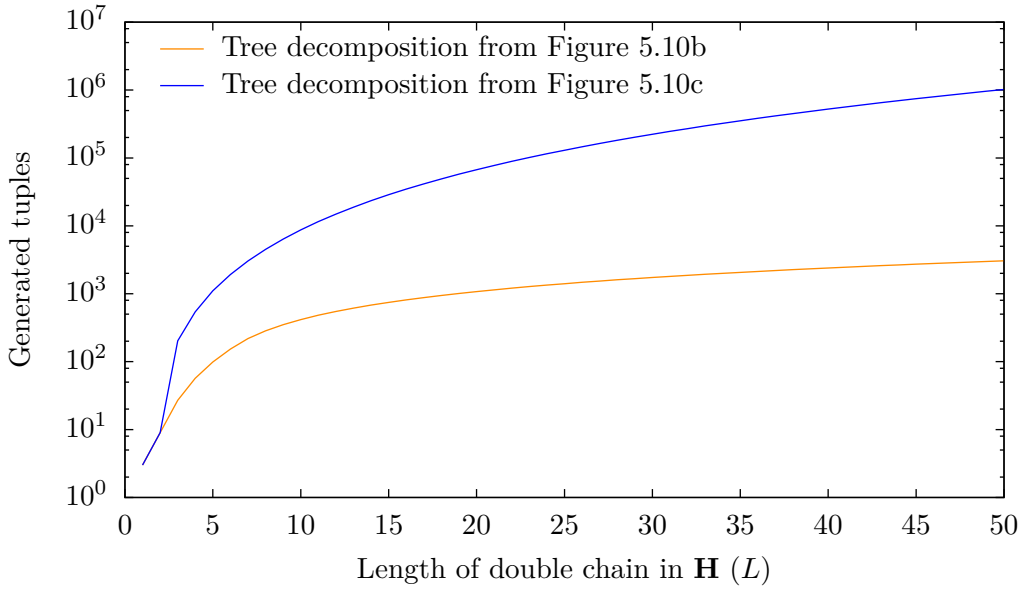


Figure 5.11: Effect of using improper tree decomposition

CHAPTER 6

Conclusion and Future work

An algorithm for graph database homomorphism problem was presented in Chapter 4. Its theoretical properties were discussed and time complexity guarantees were given. Experiments have shown that our algorithm performs better than Neo4j on large query graphs with low treewidth. A bug in Neo4j database have made some experiments inconclusive — these experiments should be repeated once this bug is fixed.

In practical applications, real query execution time is an important measure of quality. In Section 5.2, we have shown that there is a lot of space to improve the runtime of our algorithm by choosing a tree decomposition properly and expanding partial solutions in a way that relations are kept as small as possible. For every instance $(\mathbf{G}, \mathbf{H})$ a plan that minimizes number of tuples generated by the database construction algorithm should be created. A good planner should make use of both structural information about the query graph $\mathbf{G}$ and statistical information about the state of the database $\mathbf{H}$ to create such plan. It is likely that solving the task exactly is impossible and heuristical planner may therefore be proposed.

Our implementation works with immutable databases that are kept in memory. Practical database management systems have to allow changes to the database and these changes have to be stored persistently. Adapting the algorithm for these new conditions and optimizing it for maximum performance is another challenge. The long term goal would be to integrate this algorithm to the query execution system of Neo4j.

APPENDIX A

Contents of DVD

A.1 Directory structure

`/2015-Karel_Horak-Diploma_Thesis.pdf`	Electronic version of the thesis
`/tex`	LaTeX sources of the thesis
`/TreedQuery`	Sources of the application

A.2 TreedQuery

We have implemented a prototype of a graph database query engine that uses the ideas presented in the thesis. An IntelliJ IDEA project is contained in the directory `/TreedQuery`. This project is decomposed into several subprojects, the most important ones include:

`./Core`	Implementation of algorithms from Chapter 4
`./ExperimentRunner`	Implementation of experiments (discussed below)
`./HostGraph`	Representation of graphs, computation of tree decompositions
`./RecordSet`	Representation of relations and consistency graph

A.2.1 Libraries and used software

- LibTW (Thomas van Dijk, Jan-Pieter van den Heuvel, Wouter Slob)
  Java library for computing tree decomposition of a graph. Copy in `./lib/libtw.jar`
  See: `http://www.treewidth.com/`

- QuickBB (Vibhav Gogate and Rina Dechter)
  Application for computing treewidth of a graph. Path to QuickBB binary has to be

specified in the `quickbb` property (e.g. by using `-Dquickbb=`<*path*> Java command line option)
See: `http://www.hlt.utdallas.edu/~vgogate/quickbb.html`

- GraphStream
  Java library for handling graphs. This library is used to generate random graphs.
  See: `http://graphstream-project.org/`

A.2.2 Experiments

Experiments are present in the package `cz.wigsoft.graphdb.experiments` available in the `ExperimentRunner` project. Neo4j database must be running on localhost in order to run these experiments.

Some of the experiments require a model to be specified. To rely on Barabàsi–Albert model, use string `"BarabasiAlbertGenerator:`<max–links>`"`. For a complete overview of models available, see `http://graphstream-project.org/doc/Generators/`.

`GScalability`	Scalability in the size of **G** (see Figure 5.3)
`GScalabilityArtificial`	Scalability in the size of **G**, artificial instances
`HScalability`	Scalability in the size of **H** (see Figures 5.4, 5.5, 5.6)
`HScalabilityTD`	Scalability in the size of **H**, TreedQuery only
`HeuristicsTest`	Effect of heuristical node expansion (see Figure 5.9)
`TDChoiceEffect`	Effect of bad tree decomposition (see Figure 5.10)
`Neo4jTest`	Identification of Neo4j bug

List of Figures

Bibliography

[1] Albert-László Barabási and Réka Albert. "Emergence of scaling in random networks". In: *science* 286.5439 (1999), pp. 509–512.

[2] Catriel Beeri et al. "On the desirability of acyclic database schemes". In: *Journal of the ACM (JACM)* 30.3 (1983), pp. 479–513.

[3] Catriel Beeri et al. "Properties of acyclic database schemes". In: *Proceedings of the thirteenth annual ACM symposium on Theory of computing*. ACM. 1981, pp. 355–362.

[4] Jean RS Blair and Barry Peyton. "An introduction to chordal graphs and clique trees". In: *Graph theory and sparse matrix computation*. Springer, 1993, pp. 1–29.

[5] Hans L Bodlaender. "A linear time algorithm for finding tree-decompositions of small treewidth". In: *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*. ACM. 1993, pp. 226–234.

[6] Hans L Bodlaender. "A tourist guide through treewidth". In: *Acta cybernetica* 11.1-2 (1994), p. 1.

[7] Víctor Dalmau, Phokion G Kolaitis and Moshe Y Vardi. "Constraint satisfaction, bounded treewidth, and finite-variable logics". In: *Principles and Practice of Constraint Programming-CP 2002*. Springer. 2002, pp. 310–326.

[8] DB-Engines. *DB-Engines Ranking*. 2015. URL: `http://db-engines.com/en/ranking/` (visited on 12/02/2015).

[9] Reinhard Diestel. "Graph theory (Graduate texts in mathematics)". In: (2005).

[10] Jörg Flum and Martin Grohe. *Parameterized Complexity Theory, volume XIV of Texts in Theoretical Computer Science. An EATCS Series*. 2006.

[11] Michael R Garey and David S Johnson. *Computers and intractability*. Vol. 29. wh freeman, 2002.

[12] Petr A Golovach et al. "Finding vertex-surjective graph homomorphisms". In: *Acta informatica* 49.6 (2012), pp. 381–394.

[13] Georg Gottlob and Stefan Szeider. "Fixed-parameter algorithms for artificial intelligence, constraint satisfaction and database problems". In: *The Computer Journal* 51.3 (2008), pp. 303–325.

[14] Martin Grohe. "The complexity of homomorphism and constraint satisfaction problems seen from the other side". In: *Journal of the ACM (JACM)* 54.1 (2007), p. 1.

[15] Pavol Hell and Jaroslav Nešetřil. "On the complexity of H-coloring". In: *Journal of Combinatorial Theory, Series B* 48.1 (1990), pp. 92–110.

[16] David Maier. *The theory of relational databases*. Vol. 11. Computer science press Rockville, 1983.

[17] Dániel Marx and Michał Pilipczuk. "Everything you always wanted to know about the parameterized complexity of Subgraph Isomorphism (but were afraid to ask)". In: *arXiv preprint arXiv:1307.2187* (2013).

[18] Jiří Matoušek and Robin Thomas. "Algorithms finding tree-decompositions of graphs". In: *Journal of Algorithms* 12.1 (1991), pp. 1–22.

[19] Abraham Silberschatz, Henry F. Korth and S. Sudarshan. *Database System Concepts*. sixth edition. McGraw-Hill, 2010.

[20] PostgreSQL Team. *History*. 2015. URL: `http://www.postgresql.org/about/history/`.

[21] Neo Technology. *Neo4j*. 2015. URL: `http://www.neo4j.com/`.

[22] Mihalis Yannakakis. "Algorithms for acyclic database schemes". In: *Proceedings of the seventh international conference on Very Large Data Bases-Volume 7*. VLDB Endowment. 1981, pp. 82–94.